#4

PATENTS OFFICE KILKENNY

Patents Office
Government Buildings
Hebron Road
Kilkenny

I HEREBY CERTIFY that annexed hereto is a true copy of documents filed in connection with the following patent application:

Application No.	S2000/0603
Date of Filing	28 July 2000
Applicant	ARIA RESEARCH, an Irish Company of Dominic Court, 41 Dominic Street, Dublin 2, Ireland.

Dated this 26 day of September 2001.

An officer authorised by the
Controller of Patents, Designs and Trademarks.

FORM NO. 1

REQUEST FOR THE GRANT OF A PATENT

PATENTS ACT 1992

S 000003

The Applicant(s) named herein hereby request(s)
[ ] the grant of a patent under Part II of the Act
[X] the grant of a short-term patent under Part III of the Act
on the basis of the information furnished hereunder.

1. Applicant(s)

ARIA RESEARCH
Dominic Court
41 Dominic Street
Dublin 2
Ireland
an Irish Company

2. Title of Invention
A data processor

3. Declaration of Priority on basis of previously filed application(s) for same invention (Sections 25 & 26)

Previous Filing Date	Country in or for which filed	Filing No.

4. Identification of Inventor(s)

Name(s) and addresse(s) of person(s) believed by the Applicant(s) to be the inventor(s)

To Follow

5. Statement of right to be granted a patent (Section 17(2) (b))

To Follow

6. Items accompanying this Request

$ 000603

(i)	[X]	prescribed filing fee (IRP 50)
(ii)	[ ]	specification containing a description and claims
	[X]	specification containing a description only
	[X]	Drawings referred to in description or claims
(iii)	[ ]	An abstract
(iv)	[ ]	Copy of previous application(s) whose priority is claimed
(v)	[ ]	Translation of previous application whose priority is claimed
(vi)	[ ]	Authorisation of Agent (this may be given at 8 below if this Request is signed by the Applicant(s))

7. Divisional Application(s)
The following information is applicable to the present application which is made under Section 24 -

Earlier Application No.
Filing Date:

8. Agent

The following is authorised to act as agent in all proceedings connected with the obtaining of a patent to which this request relates and in relation to any patent granted -

Name & Address

Cruickshank & Co. at their address recorded for the time being in the Register of Patent Agents is hereby appointed Agents and address for service, presently 1 Holles Street, Dublin 2.

9. Address for service (if different from that at 8)

Signed Cruickshank & Co.

By:- Catherine Reedy Executive.

Agents for the Applicant

Date July 28, 2000.

"A Data Processor"

Introduction

The present invention relates to a data processor and in particular to a data processor of the Reduced Instruction Set Computer (RISC) type data processor.

As the computational requirements of data processing increase the datapath widths of the processors have correspondingly tended to increase. Typically, currently used data processors are 16-bit, 32-bit and 64-bit processors i.e. having datapath widths of 16-bit 32-bit and 64-bit datapaths. Further the number of registers within the data processors have increased not alone in size because of the datapath width, but also in number because of the complexity and of the computations.

Essentially when a data processor is being designed, the first thing that happens is that the various computational and other requirements of the processor are specified in a program. Then the designer, or programmer specifies the requirements of the data processor to tackle this task, specifying the number of bits of datapath width required, the number of registers, memory and other computational requirements. Such a processor, which will contain at least a configurable logic unit, a plurality of registers and accessible memory and the various datapaths between the components. Having done this the programmer will then choose some processor and will then specify that processor which will then be embodied in silicon. The first problem that arises for the designer is that very often he or she has to make a choice between a 32-bit processor and a 64-bit processor. Suppose, for example, the requirement is actually for something with a 37-bit datapath width, 10 registers and a certain memory and logical unit capacity. The designer has a first choice as to whether he or she will choose a 32-bit dataprocessor and use it, or a 64-bit data processor. If a 32-bit data processor is used, then obviously it will be slower than a 64-bit data processor, but almost certainly the latter will cost substantially more and the chip embodying the processor will also be substantially larger in size, probably of the order of 30%. If then the only processor that the designer can get is one with an excess capacity of registers, then the chip being purchased will also have a considerable amount of redundant space.

Further problems arise with the increase in datapath width in that the registers within the processors have also increased to have matching widths and many data being processed will be smaller than the datapath widths and thus large registers are used to store words in a wasteful manner. This is particularly the case with pure load/store machines such as implement a pipeline architecture. At the same time it is appreciated that the greater the number of dataprocessing registers available, then the more data can be stored in registers with fewer reads from or writes to cache or main memory. The disadvantage of providing a larger number of registers is the complexity and costs increase and as mentioned already increasing the size of the registers enabling them to store or manipulate a larger amount of data has the resultant disadvantages of cost, increased complexity and physical size.

RISC pipelining architecture has in general produced an increase in speed of processing to one command per processor system clock cycle. This has been achieved by replacing a micro-program in the processor by hardware and by extensive pipelining measures. One particular model, known as the Harvard model, is used in such processors and has in many instances replaced the previously used von Neumann model, which has been largely dispensed with. In the Harvard model the storage areas are separated and accessed by using different access routes. In both of these cases processing and result sequencing of the command flow is carried out.

Generally it has been realised that what is required is a processor that could be effectively infinitely configurable. It will be appreciated that practically not every component of the processor needs to be infinitely variable. At the same time there may be a need for example to have an instruction greater than 32-bits wide even if in practice at present it is practically never required. Various attempts have been made to do this, but heretofore have been relatively unsuccessful. Essentially what is required is a processor that can be specified exactly down to all the various components, whether they be the datapath width, memory, number of registers, size of registers and so on so that a designer can specify exactly the size and configuration of chip required to carry out the particular processing tasks. Thus, what is required is a processor with no redundant components either in number or size.

For example, U.S. Patent Specification No. 6,061,367 (Siemens) discloses a processor having a pipeline architecture and a configurable logic unit. This processor includes as well as the configurable logic unit, an instruction memory, a decoder unit, an interface device, a programmable structure buffer, an integer/address instruction buffer and a multiplex-controlled s-paradigm unit linking contents of an integer register file to a functional unit with programmable structures and having a large number of data links connected by multiplexers. The s-paradigm unit has a programmable hardware structure for dynamic reconfiguration/programming while the program is running. The functional unit has a plurality of arithmetic units for arithmetic and/or logic linking of two operands on two input buses to produce a result on an output bus, a plurality of compare units having two input buses and one output bit, a plurality of multiplexers having a plurality of input buses and one or two output buses and being provided between the arithmetic units, the compare units and the register file, and a plurality of demultiplexers having one input bit and a plurality of output bits. A method is also provided for high-speed calculation with pipelining.

Various other attempts have been made to provide improved constructions of processors, such as, for example, those produced by the company Arm Limited. Typical examples of their processors are described in various U.S. Patent Specifications. For example, U.S. Patent Specification No. 5,969,975 (Arm) attempts to overcome the disadvantages of the complexity and increase in number of registers by providing an arithmetic logic unit to receive input operands from M X-bit registers to produce output datawords stored within N Y-bit registers, where $M/N = 3$, $8 \leq Y-X \leq 16$ and $3X=2Y$. It is suggested that this arrangement is particularly suitable for digital signal and processing and in situations where each input operand is used a plurality of times before a new input operand is loaded in its place in a register.

U.S. Patent Specification No. 5,881,259 (Arm) is directed to accessing a memory having a plurality of memory locations for storing data values and in particular to a data processor that prevents memory access.

U.S. Patent Specification No. 6,021,476 (Arm) again is directed towards the accessing of memory in data processors.

U.S. Patent Specification No. 5,961,633 (Arm) provides a data processor in which successive data processing instructions are again executed in a pipeline architecture. This processor contains conditional control means for preventing complete execution of a current instruction if either the memory detects that a memory access initiated by a preceding instruction is invalid, or if in some way it detects that the current instructions should not be executed.

U.S. Patent Specification No. 5,132,898 (Mitsubishi Denki Kabushiki Kaisha) describes another type of processor for carrying out operations between operands having different bit lengths of data and it illustrates very clearly the problems involved in the manipulation of such data.

While various attempts have been made, as mentioned already, to provide configurable processors, a considerable amount of the activity involved has been in improving the operation of processors generally and in improving their architecture without in fact tackling the major problem which is that what the user wants is a processor directed entirely towards the task in hand i.e. to allow the programmer or designer produce a processor, which processor will have a specification ideally matched to the processing requirements. Once this has been done then a considerable amount of the problems in relation to actual processing operations, etc. become irrelevant.

Statements of Invention

According to the invention there is provided a processor having a number of components including at least a configurable logic unit, a plurality of registers, an accessible memory, and datapaths between the components, characterised in that:

the datapath width is of variable bit size namely n bits;

the number of the components are arbitrarily chosen;

where appropriate the components are of n bit size; and

each component is programmed to handle data having one of two sizes

$\leq n$ or $> n$.

While the number of components is arbitrarily chosen, this is largely done for ease of designing the processor, but the processor essentially could have a components, where a was any number. It has been found in practice for example that producing 32 registers in the actual design of processor is more than enough registers for any particular use. There is however no reason why additional registers could not be produced and prepared. Many of the components will have to be of the n bit size to match the datapath width, but other of the components need not. For example, it is envisaged that the registers can be specified to any bit size, thus overcoming the problems as mentioned already in relation to register sizes. It is however important to appreciate that any input can be greater than or less than the datapath width size. Particularly this may be the case with memory where memory sizes will be larger than the datapath width. Further it may be that there will be situations that a datapath width, for example, of 150-bits might be required in rare situations, but generally what would normally be required would be 73-bits, therefore the programmer would chose a 73-bit datapath width and would so-program it that the 150-bit wide word could be handled when required.

Ideally such a processor comprises:

means to select the number and size of each component;

means to select the datapath width;

means to configure the components for that datapath width; and

means to compare the width of a data input to the selected datapath width that has been chosen for the component.

By having such a processor, once the programmer has specified the requirements, it

is possible for the programmer then to simply take the processor according to the present invention and input the various data. Then having inputted the various data requirements, such as, for example, in a database or other document, the processor can be used to effectively provide the processor and make it in silicon.

It will be appreciated therefore to a certain extent what is being provided according to the present invention is not so much a processor, but in fact a template to allow a processor to be produced, in the sense that there will never be produced a processor of n bits wide. What will be produced for example is a processor with a datapath width of 37 with for example 15 registers, a configurable logic unit containing the logic required and accessible memory. This will then be manufactured in silicon, which will also mean that it will be as quick as using a standard 64-bit processor and only marginally more bulky and costly than a 32-bit processor. If fewer registers were requirement it might be less costly and bulky than an off-the-shelf 32-bit processor.

In one embodiment of the invention when the immediate value of an instruction is limited in size to a preset number of bits and when the total instruction size is less than n the instruction is expanded to n bits wide. However, when the immediate value of each instruction means that the instruction size becomes greater than n, then the instruction has to be truncated or compressed in the sense that the programmer is required to specify and ensure that the immediate value can be handled by the processor.

Ideally the processor has special registers essential to the operation of the processor and then general registers. Very often some of the special registers will have a pre-set size which cannot be altered. The general registers, however, are entirely dependent on the bit size of the data being handled and many of them will in fact be of size n bits, but not all of them need to be.

It is envisaged that one or more of the general registers may be mounted external of the processor and the processor according to the invention is so-configured and thus all that a designer requires is to specify those registers to be held external.

In a particular embodiment of the invention the special registers are programmed to

allow their content to be written to memory external of the processor and then to enable the functionality of a general register. In this way in certain situations the special registers can have two functions, which further reduces the size of the processor. They will act as general registers when required and will still be able to act as special registers.

In many instances all the general registers will indeed be n bits wide.

It is to be appreciated that the most significant bit (MSB) of the data is the n^{th} bit and this becomes our sign bit. For example, in the processor means are provided for writing the n^{th} bit of data to an address of size (x-bits) greater than n bits, the bits in positions x–n are populated by the MSB. Similarly, for example, if we had the processor of 23-bits and we had an immediate which is 16-bits long it could be loaded into the processor and a sign indication bit (namely bit 15) could be extended a variable length to match the datapath width, generally the extension is by n-x. Thus, the MSB would simply be extended unto the end of the datapath width.

When it is required to compare data in the high order address of one word with data in the low order address of another word when n is an even number means are provided to compare the data in the top half of one word with the data in the bottom half of the other word ignoring the remainder of the data. For example, when it is required to perform logical operations comprising bitwise AND, OR and Exclusive OR on data in the high order address of one word, with data in the low order address of another word when n is an even number, then the means are provided to compare the data in the top half of one word with the data in the bottom half of the other word while ignoring the remainder of the data. This is done by swapping the top and bottom half of the first operand data prior to performing the operation. This operation which is known as ANDX, ORX and XORX depending on the logical operation involved and can involve either registers or immediate operands. The S operation as mentioned already has a result in the top and bottom halves which are swapped. This operation is indicated by postfixing the operations AND, OR and XOR with an S resulting in ANDS, ORS and XORS. These operations are achieved through the cross wiring of the top and bottom halves of the first operand and also of the result and a multiplexing arrangement selected by the operation itself which

delivers the correctly swapped data to the appropriate location.

The processor according to the invention is designed and arranged so that separate logic gates to perform logic operations can be provided in separate units which can be plugged into the processor by hardwiring. These would largely be in relation to specific operations, such as, for example, the encryption of a certain amount of data at rare intervals. The encryption of the data would slow down the whole processor and thus it is extremely advantageous to take that one operation into a hardwired logic processor and keep it entirely separate for the remainder of the processor and simply access it when required which can all be done within one clock cycle rather than the thirty or forty clock cycles it would take to carry out the logic operation within the processor itself.

The processor according to the present invention can be so-arranged that both registers can be shared between various processors. Thus, for example, in certain situations when more than one processor would be required in a particular application in the sense that while the designer or programmer might require two or more processors, that in the specifying of those processors it would be possible to use registers in one processor for tasks in the other processor.

It is envisaged that the processor according to the present invention will be embodied in a computer disk or the like storage medium and can be simply downloaded by an operator, the various parameters inputted, the processor configured and then downloaded for subsequent manufacture in silicon or the like material.

Further the invention provides a method of designing a processor comprising the steps of:

preparing an outline processor in general architecture having a series of components described by blocks or the like interconnected by various datapaths having at least a configurable logic unit, a plurality of registers, an accessible memory, and such other units and components as are required for a processor of the type being designed and then defining the datapath width of variable bit size namely n bits;

choosing an arbitrary number of components greater than that which would ever be required such as, for example, 64 registers; or alternatively

choosing a component where a is any number that could be chosen; defining the components size as n bit size; and

programming each component to handle data having one of two sizes, namely $\leq n$ or $> n$.

In this way a general processor is designed and then subsequently when it is required to produce a processor from this general design the number of components, the datapath width size and so on are chosen and they are entered into a database, which database will allow a particular design of processor to be produced.

Detailed Description of the Invention

The invention will be more clearly understood from the following description of an embodiment thereof given by way of example only with reference to the accompanying drawings, in which:

Fig. 1 is a block diagram of a processor according to the invention and the external interfacing;

Fig. 2 illustrates the basic processor pipeline;

Fig. 3 illustrates the basic processor pipeline with control signals;

Fig. 4 illustrates the stall mechanism;

Fig. 5 illustrates the forwards for data hazards;

Fig. 6 illsutrates the processor pipeline in more detail;

Fig. 7 is a block diagram of the processor information;

Fig. 8 is a flow diagram of the instruction information unit;

Fig. 9 is a block diagram illustrating the sequencer;

Fig. 10 is a flow diagram illustrating the forwarding decision logic;

Fig. 11 is a block diagram of comparator logic;

Fig. 12 is a flow diagram illustrating the decoder;

Fig. 13 is another flow diagram illustrating the decoder;

Fig. 14 is a flow chart illustrating R-type instructions;

Fig. 15 is another flow chart illustrating the R-type instructions;

Fig. 16 is a flow chart illustrating I-type instructions;

Fig. 17 is another flow chart illustrating the I-type instructions;

Fig. 18 is a flow chart illustrating the I-type (load) instructions;

Fig. 19 is another flow chart illustrating the I-type (load) instructions;

Fig. 20 is a flow chart illustrating the I-type (store) instructions;

Fig. 21 is another flow chart illustrating the I-type (store) instructions;

Fig. 22 is a flow chart illustrating jump instruction;

Fig. 23 is a flow chart illustrating JAL instruction;

Fig. 24 is a flow chart illustrating RET instruction;

Fig. 25 is a flow chart illustrating JR instruction;

Fig. 26 is a flow chart illustrating JALR instruction;

Fig. 27 is a flow chart illustrating BEQZ and BNEZ instruction;

Fig. 28 is a flow chart illustrating HLI instruction;

Fig. 29 is a flow chart illustrating HALT instruction;

Fig. 30 is a flow chart illustrating TRAP instruction;

Fig. 31 is a flow chart illustrating RFE instruction;

Fig. 32 is a flow chart illustrating illegal instruction;

Fig. 33 is a flow chart illustrating register selector;

Fig. 34 is a flow diagram illustrating stall controller;

Fig. 35 is a block diagram of the execution unit;

Fig. 36 is a block diagram of the arithmetic and logic unit (ALU);

Fig. 37 is a flow diagram of the adder unit control logic;

Fig. 38 is a diagram of the adder unit within the ALU;

Fig. 39 is a flow diagram of the logic unit;

Fig. 40 is a flow diagram of the replication logic;

Fig. 41 is a block diagram of the multiplexers with SourceOperandMuxes;

Fig. 42 illustrates the control and selection of output result from the execution unit;

Fig. 43 is a flow diagram illustrating the data unit information;

Fig. 44 is a flow diagram illustrating the data memory controller;

Fig. 45 is a flow diagram illustrating the data memory sign extend unit;

Fig. 46 is an overall block diagram of the register unit;

Fig. 47 is a block diagram of the general purpose registers; and

Fig. 48 is a block diagram of the register multiplexers (muxes).

Referring now to Fig. 1 there is illustrated in block diagrammatic form an outline of the processor according to the invention and the external interfacing to it. All of the external interfacing has various signals to and from the processor. The processor is identified by the reference numeral 1 and the principal components illustrated are instruction decoding 2 which in turn feed an arithmetic logic unit (ALU) 4 through datapaths 5 of n bits wide. Further datapaths 5 are also illustrated as is a data memory control 6 fed from the arithmetic logic unit 4. The data memory control 6 also feeds the general purpose and special registers which together with the instruction decoding 2 also feed the arithmetic logic unit 4 through a
Signal descriptions for Fig. 1 are listed below and are elaborated on somewhat later.

sysClk

This is the system wide clock provided to the processor. All pipelining and registering within the processor is done on this clock.

sysReset

This is the reset signal provided by the system. It is active low.

imAddr[m-1:0]

This is the instruction memory address bus. It is a registered output. It is in byte address sizes but all values that appear on it are word addresses. On a reset this bus goes to zero.

imData[p-1:0]

This is the data from the instruction memory i.e. it is the instruction addressed by the instruction memory address bus.

imRdy

This signal indicates when valid data is available from the instruction memory. If the instruction memory takes more than a clock cycle to produce valid data from when it is addressed, this signal must be pulled low until valid data is available.

dmAddr[q-1:0]

During accesses to data memory, the address of the data location appears on this bus. It is a registered output. The addresses that appear on this bus are byte addresses.

dmDataIn[n-1:0]

If a Load from memory instruction occurs, the data from the data memory location, addressed by *dmAddr[q-1:0]*, is passed to the processor on this bus.

dmDataOut[n-1:0]

If a Store to memory instruction occurs, the data to be written to the data memory location, addressed by *dmAddr[q-1:0]*, appears on this bus. This is a registered output.

dmCS

When this signal is HIGH it indicates that an access to memory is occurring

dmRW

This signal indicates to the memory whether a load or store is happening. If it is HIGH this indicates a store to memory and if it is LOW a load from memory is happening.

dmSiz[1:0]

This output signal is used by the processor to indicate to the data memory when word, halfword or byte transfers are required. **b00** indicates that the transfer is a byte, **b01** indicates that the transfer is a halfword and **b10** indicates that the transfer is a word. These values are valid for both loads and stores.

dmRdy

This input signal indicates when valid data is available from the data memory. If the data memory takes more than a clock cycle to produce valid data from when it is addressed, this signal must be pulled low until valid data is available.

extInt

This input signal is the request from an external device to interrupt the processor. It must be held high for at least 1 *sysClk* clock cycle.

extIntAck

When the processor receives the external interrupt and starts to service the interrupt, this signal is set high for 1 *sysClk* clock cycle to acknowledge the interrupting source that it has received the interrupt.

As explained the architecture of the processor is based around the Harvard architecture model. This model includes the non-sharing of instruction and data memory space which lends itself to a very low cycle per instruction count as there is no contention for memory. Potentially if there is zero wait memory, such as asynchronous SRAM, the processor will not have to stall and wait for any memory access to be completed. Essentially the processor according to the present invention is shown in five stages. This is illustrated in Fig. 2

The pipelining technique allows the overlapped execution of multiple instructions. The pipeline in the present processor is divided into five stages. All of the stages

use the same clock cycle so an instruction is completed every clock cycle and the duration of an instruction is five clock cycles. It will be appreciated therefore that this is a particularly suitable form of processor as the through-put is increased by a factor of five, under ideal conditions. It is important to appreciate that all the stages are active on every clock cycle. All stages have to complete in one clock cycle and any combination of instructions is programmed to occur at once.

Referring now to Fig. 2 the elements of each stage of the pipeline is described in somewhat more detail with the memory connected thereto. The processor is again indicated by the reference numeral 1 and the stages are divided into five stages, namely a Fetch stage 10, a Decode stage 10, an Execution stage 30, a Load and Storage stage 40 and a Write Back stage 50. Because the stages are identified by different reference numerals, the components previously identified by a reference numeral now may have a different reference numeral attached thereto. The Fetch stage 10 implements the loading of the next instruction to be executed. A program counter (PC) keeps track of the instruction number to be executed. The Fetch stage 10 includes an instruction memory 11 and address buses connected to this instruction memory 11 and a multiplexer (mux) 12 to select the next PC. The memory is addressed by the actual value of PC and the content of that position is registered and sent to the decode stage 20. The multiplexer 12 selecting the next PC is dependent on the instruction being decoded at the same clock cycle in the decode stage. It determines whether to choose from the PC +4 or the target address for branch or jump instructions. It is passed out as the instruction memory address and the data returned. In the Decode stage 20 after the instruction has been passed from memory the Decode stage 20 decodes the instruction to determine the operation to be performed in operands that are selected by the instruction. These operands are from registered address by the instruction, or a value provided by the instruction. This is where the whole control of the whole pipeline occurs. It takes the instruction from the Fetch stage 10 and decodes it in order to set the signals which will control its execution. Part of the information present in the instruction being decoded are the addresses of the registers involved in some operations. There is thus provided a decoder 21, general purpose and special purpose registers 22, a sign extend unit 23 and a multiplexer 24 for selecting the next PC. Part of the information present in the instruction which is being decoded

in the decoder 21 are the addresses of the registers 22, thus they address the source operands of the general purpose and special registers 22 and their contents are registered to become the inputs for the Execution stage 30. In the case of an immediate operation, the 16-bit immediate value, which is the usual value coming from the instruction is either sign extended or padded with zeroes in the sign extender 23. This sign extend unit 23 is a dedicated sign extend unit that will be described in some more detail below. Generally speaking the instruction data will be in 32-bits with 16-bit immediate value. The processor can be designed for higher inputs, but they are not generally required and thus in the description of the processor there is this limitation.

When decoding a branch instruction or a jump, the value of next PC is appropriately changed. Decisions of whether to change or not to change the value of the PC and the calculation of the target address are done in the Decode stage 20 by means of control logic and an additional adder. In the case of TRAP, RET or RFE instructions the PC is also changed from the normal flow to a predetermined value. Again this is reasonably standard.

The Execution stage 30 is where the actual implementation of the operation decoded in the Decode stage 20 is performed. This is where an ALU unit 31 is illustrated which ALU unit 31 is in fact the ALU 4 already identified in Fig. 1. In the Execution stage 30 the ALU operation indicated in the instructions and registers is performed and delivered to the next stage of the pipeline. It calculates the address for the data memory access in the Load/Store stage 40 which will be performed in the next clock cycle in the Load/Store stage 40. The source operand could be either a register or an immediate. Thus, there is a multiplexer 32 provided to decided between them.

The next stage is the Load/Store stage that also could be called the memory stage 40. The data memory address 41 and data buses, as well as the corresponding control signals in turn has a further multiplexer 42 to all the either the memory data or ALU result to be registered in what is effectively the last stage, which is the Write Back stage 50. The Write Back stage 50 has no specific hardware, it only passes

data to be written to the general purpose registers or special purpose registers 22 and the control signal to do it.

The above is a brief outline of the architecture. It does not describe it in great detail and indeed most of the architecture can be said to be essentially conventional, except for some minor features.

However, the processor according to the present invention is extensively configurable and parameterizable. The datapath width has been set at n bits and the number of registers and the size of instruction and data memories is configurable.

The data length of the datapath elements and almost all the registers of the processor can be configured to any width from 1 bit to n bits, namely a word length of n bits for the processor. The processor according to the invention also uses data of two other sizes, namely halfwords which are half the width of the word length, needless to say if the word length is an odd length, namely n is not an even number, the half word is modulus of half the width of the word length. Sometimes in the following discussion bytes which are 8-bits is used, but will be understood by those skilled in the art.

According to the invention the width and amount of registers within the processor may be configured. Again this is described in more detail. For ease of design and use, it is normal to pick a maximum number of registers according to the invention, such as, for example, 64 registers and to design the processor for 64 registers. Needless to say greater numbers of registers may be provided, however the computational work to handle such registers means that they are not always necessary for every processor. Thus, generally speaking the registers, except for some are of n bits wide and the actual number of registers is arbitrarily chosen in due course as will be explained later from what are effectively a fixed number of resisters. The important point to appreciate is that all these registers are provided which may be used as required.

In the particular processor according to the present invention the instruction and

data memories are physically outside of the processor, however, the amount of memory accessible is defined by the processor. Both the instruction memory address, *imAddr*, and the data memory address, *dmAddr*, are generated by the processor and the width of these busses can be set to match the size of memory needed (see Fig. 1).

The data width of these memories can also be configured with the data memory width matching the width of the processor datapath width, namely n.

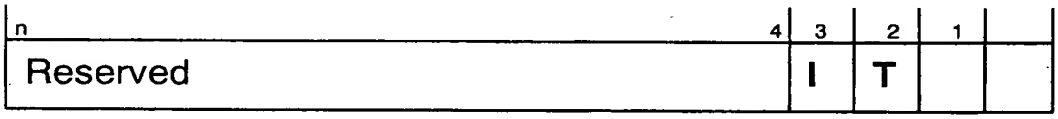
In relation to the instruction memory while this instruction memory width is determined by the width of the instructions, the processor according to the present invention is so-arranged that the instruction memory width can be variable. However, at the present moment because it has been found that relatively seldom is an instruction memory width greater than 32-bits required, the present design has been carried out with the instruction memory with width fixed at 32-bits wide. Obviously this has the ability to be changed if the instruction memory use some form of compression or alternatively could be extended.

Various other configurations of the processor are included, but these are simply program configurations such as, for example, interrupts can be enabled or disabled and special hardware functions can be added as special ALU operations. Again this in more detail below. Further instructions which are derived from the instructions of the processor can also be added.

Reference has already been made to the registers and they have been described as both general purpose registers and special registers. The general purpose registers (GPR) are the set of registers that can be read or written to by all instructions that access registers. Usually register R0 contains 0 and cannot be written to. The number of GPRs in the processor can be configured in this particular embodiment up to a maximum of 32 and the width of the registers match the configured datapath width n. Needless to say this is one particular example of the invention, but it has been found for ease of programming better to fix on a certain number of registers and reduce the number of registers when required in the actual processor produced, rather than simply define the registers by another indeterminate number.

The way in which the datapath width n is determined. The fact that the number of registers is fixed does not in any way impinge on the configurability of the processor. It would be possible to very simply design a processor according to the invention with 100 registers and then subsequently in use simply reduce the number of registers required.

The special registers are a second set of registers in the processors. These registers can be read or written by instructions that perform an operation where the two source operands are register values. The first four registers are used for controlling the processors and these four registers are a reason register, link address register, exception address register and an interrupt address register. It is possible to configure up to 32 registers as with the GPRs. The reason register explains the present state of the processor 4 bits are used and generally they are as listed below.



where:

R – indicates that the processor has been powered up from a full hard system reset.

O – If the processor received an illegal instruction this bit will be set and the processor will start executing from the start address again. It also will have the effect of clearing the R bit so that it is indicated that the last reset was a soft reset as opposed to a hard system reset. If this bit has been set and there then is a hard reset, this bit will be cleared.

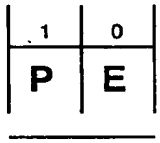
T – If a trap instruction has been encountered, the processor will execute an exception service routine and while this routine is being executed, this bit is set. On exiting the exception service routine, this bit will be cleared.

I – This has the same operation as the T bit except it is set while an exception service routine is being executed as a result of an external interrupt.

The link address register contains the value of the return address when the code being executed jumps to another instruction and intends to return back to the original section of code. An example of this is a procedure call. The width of this register has a minimum size of the instruction memory address width a if the datapath width n is less than a however, if the datapath width is greater than the instruction memory address width (a), this register takes on the size of the datapath n.

The exception address register is at register address 3 in the special registers set. This register contains the value of the return address when the code being executed jumps to another instruction and intends to return back to the original section of code. The instructions that cause it are JAL and JALR. If those instructions are not present in the code, this register can be used as Special Register otherwise the return address will be overwritten.

The interrupt register is at register address 0 in the special registers set. It is a 2-bit register and only the lower bit which indicates if the interrupt enabling can be written.



This register and support logic controls the interrupt handling of the processor. When an interrupt is received the pending bit P is set. Then if the enable bit E is set, the processor will automatically service the interrupt. When the register is read the bit P appears in the bottom bit of the upper half of the word and the bit E in bit 0. The rest of the bits are zero.

There are four instruction formats shown below in the table. As all the opcodes have not been used, many more additional instructions may be introduced.

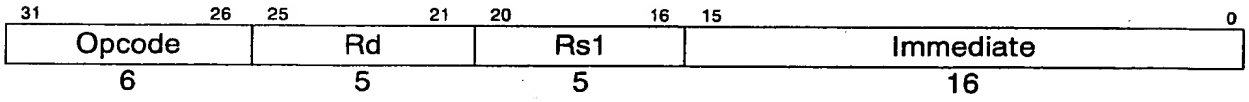
In the present processor the instructions have initially been implemented at 32-bit wide. However, this has been set as a parameter of bit wide p, which can be changed if a reduction in instruction memory width is required and some form of Fetch stage decompression is used to expand the instruction to its intended size.

The first format is an I-type (immediate) instructions which manipulates data provided by a 16-bit signed or unsigned immediate field. These immediate instructions break down as follows:

- Immediate ALU operations where the immediate is used as an operand for the ALU and the result is written back to a register.

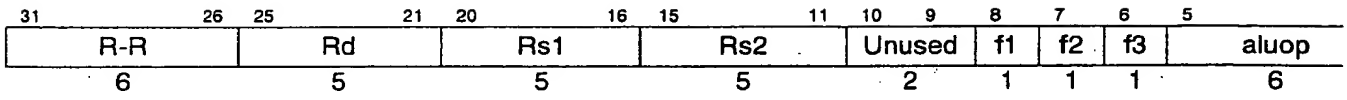
- Conditional branch instructions where the immediate is added as an *offset* to the Program Counter to transfer control of the processor to a different point in the source code.

- Load from Memory and Store to Memory instructions use the immediate data as the offset to a register value to generate the memory address to be accessed.



The second format of instruction is the R-type (register to register) instructions which perform pure ALU type operations on two operands provided by two source registers specified in the instruction. The result is always destined for a register. The operation to be performed is specified by the aluop field of the instruction. Access to the Special Register set from source code can only happen through R-type instructions except for Special Register 1 (Link Address Register). Special Registers are identified by 3 1-bit flags in the instruction and are shown and

explained below.



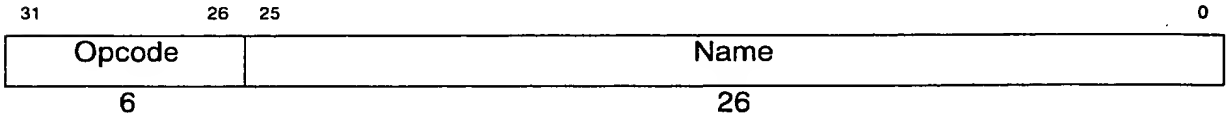
f1 = 0 => rs1 is addressed in the General Purpose Registers;
f1 = 1 => rs1 is addressed in the Special Registers;

f2 = 0 => rs2 is addressed in the General Purpose Registers;
f2 = 1 => rs2 is addressed in the Special Registers;

f3 = 0 => rd is addressed in the General Purpose Registers;
f3 = 1 => rd is addressed in the Special Registers.

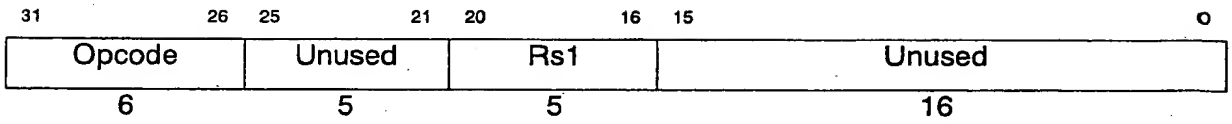
The third type of instruction is the J-type (jump) instructions which are the unconditional jumps in source code transfer control. There are 4 instructions grouped in this type, Jump, Jump Register, Jump And Link and Jump And Link Register. The two Jump And Link based instructions retain the next instruction address from the jump instruction so that program control can return to the point the jump was executed. This address is stored in the Link Address Register in the Special Register set.

This following is the make up of the Jump and Jump And Link instructions. A 26-bit *name* is sign extended and added to the Program Counter to create the address of the next targeted instruction



The Jump Register and Jump And Link Register instructions are constructed as follows, where rs1 is the General Purpose register address whose contents is the

address of the targeted instruction.



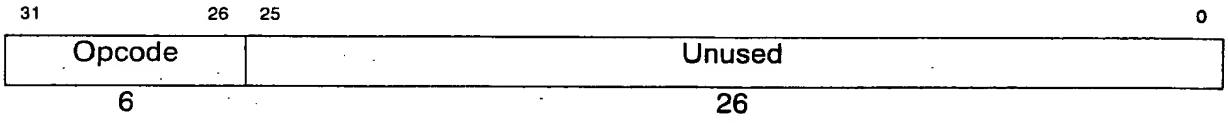
The fourth type of instruction is C-type (control) instruction which is used for processor control type functions. They contain a simple opcode with no register or immediate referenced.

The HALT instruction will stall the EVE Processor pipeline and continued operation will not commence until an interrupt is received.

The RET instruction transfers control back to the section of code jumped from by a JAL or JALR instructions.

The TRAP instruction is a mechanism for allowing software to transfer from the main code to the Exception Service Routine.

The RFE instruction returns control from the Exception Service Routine back to the main code after either a TRAP instruction or an interrupt has been serviced.



In most cases namely R type instructions the use of rs1,rs2 and rd is very clear. For example, it could be

add r5,r4,r3 => rd = r5; rs1 = r4; rs2 = r3

However in the case of I type, namely Load and Store type instructions this is not that clear and thus it is necessary to be aware that:

For a STORE e.g. SW offset(R10), R3 rd = R3, rs1 = R10, immediate = offset
For a LOAD e.g. LW R3, offset(R10) rd = R3, rs1 = R10, immediate = offset

Also for Jumps and Branches based on register values(BEQZ, BNEZ, JR, JALR) the register is in rs1 not rd i.e. in bits 21:16 of the instruction word which means rd (bits 25:21) should be zero.

As mentioned already, it is envisaged that many more instructions may be introduced and the processor according to the invention is adapted to such further types of instructions as defined, but have not been assigned opcodes in this processor, but can be included as required by the programmer

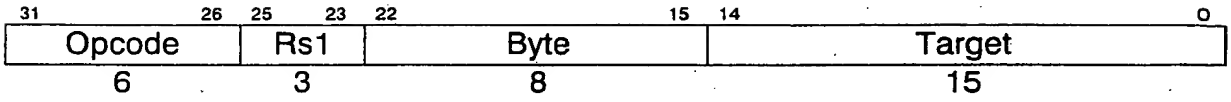
These instructions implement a branch conditioned on the comparison between the selected byte, halfword or word specified in a register and the corresponding byte, halfword or word specified by the immediate (only for bytes) or in another register. These instructions are implemented in two different formats one for the byte immediate branches and the branch on register compare.

These yield in 22 new branch instructions:

Table 2 – Branch Instructions

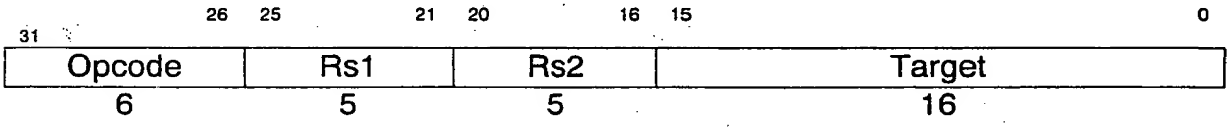
No. of Opcodes	Mnemonic	Instruction meaning
4	BEQBxI	Branch if byte x equal to byte immediate
4	BNEBxI	Branch if byte x not equal to byte immediate
4	BEQBx	Branch if byte x equal to byte register
4	BNEBx	Branch if byte x not equal to byte register
1	BEQHU	Branch if upper half equal to half register
1	BNEHU	Branch if upper half not equal to half register
1	BEQHL	Branch if lower half equal to half register
1	BNEHL	Branch if lower half not equal to half register
1	BEQW	Branch if word equal to register
1	BNEW	Branch if word not equal to register

The format of the BEQBxI and BNEBxI instructions is as follows:



When these instructions are executed, the immediate byte in the instruction is compare to a byte in the data store in a register pointed to by Rs1. This register address field is only 3 bits in size, therefore, the byte can only be compared to the contents of one of the first 8 GPRs. If the comparison is TRUE, the 15 bit Target is added to the contents of the PC and used as the next address.

The format of the remainder of the Branch on value compare instructions is as follows:



Here, the values contained with in the two registers, addressed by the instruction, are compared and if this compare is true, the Target is added to the current PC and used as the next address.

ALU instructions such as the Adds and Shifts can be implemented to use a carry set by the execution of the previous instruction to affect the carry. Although these are not fully specified, the capacity to implement them is available.

For the Add operation in the ALU, the previous carry is added along with the two source data.

For Shift operations, the previous carry is shift in to the end of the data being shifted and the bit falling of the end is stored as the next carry bit.

The carry bit can also be used to branch on. In executing this instruction, the branch will be taken if the carry is Set or Clear depending on the type of test specified.

The following Table gives a list of the instructions.

Table 2 – List of Instructions

Mnemonic	Instruction meaning	type	OpCode	AluOp
Data transfers:				
LB	Load byte	I-type	100000	
LH	Load half word	I-type	100001	
LW	Load word	I-type	100011	
LBU	Load byte unsigned	I-type	100100	
LHU	Load half word unsigned	I-type	100101	
SB	Store byte	I-type	101000	
SH	Store half word	I-type	101001	
SW	Store word	I-type	101011	
Arithmetic/logical:				
ADD	Signed addition	R-type	000000	000000
ADDI	Signed immediate addition	I-type	001000	
ADDUI	Unsigned immediate addition	I-type	001001	
SUB	Signed subtraction	R-type	000000	000001
SUBUI	Unsigned immediate subtraction	I-type	001010	
AND	And	R-type	000000	000100
ANDI	And immediate	I-type	001100	
OR	Or	R-type	000000	000101
ORI	Or immediate	I-type	001101	
XOR	Xor	R-type	000000	000110
XORI	Xor immediate	I-type	001110	
LHI	Load high immediate	I-type	001111	
SLL	Shift left logical	R-type	000000	001100
SLLI	Shift left logical immediate	I-type	010100	
SRL	Shift right logical	R-type	000000	001101
SRLI	Shift right logical immediate	I-type	010110	
SRA	Shift right arithmetic	R-type	000000	001110
SRAI	Shift right arithmetic immediate	I-type	010111	
SEQ	Set equal	R-type	000000	000011
SEQI	Set equal immediate	I-type	011100	
SLT	Signed set less than	R-type	000000	001000
SLTI	Signed set less than immediate	I-type	011000	
SLTU	Unsigned set less than	R-type	000000	001001
SLTUI	Unsigned set less than immediate	I-type	011001	
SLE	Signed set less than or equal to	R-type	000000	001010
SLEI	Signed set less than or equal to immediate	I-type	011010	

SLEU	Unsigned set less than or equal to	R-type	000000	001011
SLEUI	Unsigned set less than or equal to immediate	I-type	011011	
Control:				
BEQZ	Branch on General Purpose Register equal to zero	I-type	000100	
BNEZ	Branch on General Purpose Register not equal to zero	I-type	000101	
J	Jump (PC relative)	J-type	000010	
JR	Jump Register (target in general purpose register)	J-type	010010	
JAL	Jump And Link (PC relative)	J-type	000011	
JALR	Jump And Link Register (target in general purpose register)	J-type	010011	
TRAP	Transfer to operating system at a vectored address	J-type	010001	
RFE	Return to user code from an exception service routine	J-type	010000	
HALT	Halt instruction execution and idle until interrupt	I-type	000111	
RET	Jump to stored return address	J-type	000001	

After the definition of the pipeline stages elements, the next step is defining the control of their functionality. Because the instructions decoded in the Decode stage are effectively executed in the Execution or Memory stages, some control signals have to be generated and adequately delayed to make them effective at the right time.

The solution adopted by the present invention architecture is sending through the pipeline the control signals along with the data, so they automatically will appear at the right clock cycle in the expected stage. The problems that arise using this configuration, like hazards and stalls, will be discussed below.

Referring to Figs. 2 and 3 all the control signals are generated in the Decode stage 20 depending on the instruction being decoded. They are sent through the pipeline in the case of being used in Execution 30, Memory 40 or Write Back 50 stages or directly used in the Fetch 10 or Decode 20 stages without being registered.

There is only one control signal used in the Fetch stage 10. It is the select signal for

the PC mux. The decision is taken in the Decode stage 20 after deciding if the PC should be just incremented by four (the normal program flow) or be loaded with a different value.

The Decode stage 20 not only generates the control signals for other stages but also generates control signals for itself. It is the case of the signal controlling the sign extend unit. The immediate value coming in the instruction is either sign extended or padded with zeroes, depending on the operation being signed or unsigned. The select signal indicates which extension has to be done.

Apart from the control signals, the Decode stage 20 sends to the Execution stage 30 the PC value. It is stored in LAR (Link Address Register) when executing jump and link instructions (JAL or JALR).

The Execution stage 30 also requires control signals for the muxes choosing the proper source operands for the ALU 31 operation and the ALU 31 needs a signal indicating which operation has to perform on them. There are also four control signals passing through this stage. They will be used in the following stages.

Two of the four control signals received by the Memory stage 40 are used on it. One enabling the data memory 41, thus indicating a load or a store instruction and the other with additional information to generate more control signals for the data memory 41. These two signals are processed in a sub-block in order to generate the RW and size signals to accordingly control the data memory operation. This block also generates a select signal for the data mux. The mux chooses the memory contents in case of performing a load instruction, otherwise the ALU 31 result is passed to the final stage. The other two control signals pass through this stage and will be used in the last one.

Despite the Write Back stage 50 does not have hardware at all, it passes the data to be written, as well as the destination register address and the RW signal to the general purpose and special registers 22.

The flow of any instruction being executed in the processor starts in the Fetch

stage, where PC addresses instruction memory 11 and the instruction is read from that position. In the next clock edge, it is registered to the Decode stage 20 where the main decisions are taken. As the result of those decisions, the proper control signals are generated and sent to allow its execution.

In the case of an R-type instruction, the first action done in the Decode stage 20 is addressing the source registers 22. The content of these registers is registered out to the Execution stage 30 along with the ALU 31 operation, the select signals for the source operand muxes, the destination register address and the write enable signal. The rest of the control signals are driven to default. The select signal for the PC mux will chose PC+4 due the program flow will normally continue.

In the Execution stage 30, the source operand muxes pass the corresponding source operands and the operation indicated by the ALU operation signal is performed on them. The result is registered to the Memory stage. This stage and the Write Back stage 50 pass the ALU result along with the address and control signals to the registers 22 to be written. It is because there is no data memory access to perform.

When performing an I-type instruction involving an ALU 31 operation, the situation is similar to the one described above, except that the second source operand is an immediate. It is extended and registered in the Decode stage 20. In the Execution stage 30 it is chosen as a second operand, instead of Rs2, by means of the select signal. If the instruction is a load or a store, Rs1 and the immediate are added to form the data memory 41 address. In the case of a store, Rs2 holds the data to be stored, so it is also passed to the Memory stage 40 and there is no destination register.

The control signals indicating the type of load or store instruction (signed or unsigned, byte, halfword or word) sent from the Decode stage 20 are processed in the Memory stage 40 to produce the proper control signals for the memory. It also generates a select signal for the mux choosing between the memory data (in case of a load instruction) or the ALU 31 result.

The store instruction is finished in the Memory stage 40, because there is no data to write back to the registers 22. Nevertheless, the Write Back stage 50 sends the data, destination register address and write enable signals to the registers 22 as usual. In this case, the data sent is the ALU 31 result and the destination register is R0 (not writable).

When the I-type instruction is a Branch the actions taken are different. The register 22 indicated in the instruction is addressed as usual, but its content is compared with zero to decide whether the branch should be taken or not. It is done in the Decode stage 20. Depending on that decision, next PC is selected adequately in the Fetch stage.

The control signals sent through the pipeline are defaulted because any actions are required further on. The Execution stage 30 thus performs an addition on the register 22 addressed and R0 and the destination address is set to R0. It is equivalent to perform a NOP which is defined as: ADD R0,R0,R0.

In the case of a branch taken, a signal is set in the Decode stage 20 to indicate that the next instruction has to be annulled. It is because that instruction was fetched while decoding the branch instruction but it should not be executed. If the branch is not taken, the program flow normally continues.

To annul an instruction means that despite it has been fetched, it will not be executed. To do so, the Decode stage 20 sends to the pipeline a NOP, ignoring the contents of the instruction.

The J-type instructions change the value of next PC unconditionally. What is decided in the Decode stage 20 is which value of next PC has to be chosen and whether to store or not the value of actual PC in order to continue the normal program flow after returning from the jump routine. These instructions cause the next instruction to be annulled.

The J instruction includes an offset to be added to the actual PC to form the target address. That address is chosen by the select signal of the PC mux as the value

for next PC. A NOP is sent to the pipeline because nothing has to be calculated in the Execution stage onwards.

JAL instruction does the same, except that actual PC is stored in the Link Address Register. When the instruction RET is found in the corresponding JAL service routine, the value stored in LAR is loaded back into PC to allow the program flow to continue.

JR and JALR address a register, which content is directly loaded as next PC. Again, in the case of a JR a NOP is sent to the pipeline and in the case of a JALR, the value of actual PC is stored in LAR.

The control transfer instructions accordingly change the value of PC. The instruction TRAP or an interrupt cause next PC to be loaded with a predetermined address and actual PC to be stored in Exception Address Register (EAR). The content of that address is either the first instruction of the Exception Service Routine (ESR) or an instruction to jump to it. RFE marks the end of the ESR and causes PC to be loaded back with the content of EAR.

RET does the same as RFE, but loading the content of LAR instead. That instruction is present at the end of the JAL and JALR service routines.
The instruction HALT causes the whole pipeline to stall until an interrupt is received. Every stage keeps doing the actions they were doing when the interrupt came, until the pipeline is released and the inputs of the stages are able to change.

Some combinations of instructions can create resource conflicts (hazards) that prevent the next instruction from executing at the corresponding clock cycle. When the hazards cannot be avoided by techniques like forwarding, they incurred in stalls, reducing the pipeline performance. There are also other sources for stalls like external devices not ready or the instruction HALT.

There are three situations causing pipeline stalls (apart from the ones caused by data and control hazards and discussed below): Instruction Memory or data memory not ready and HALT instruction. When any of them occur, the whole

pipeline is stalled, which means that Fetch 10, Decode 20, Execution 30 and Memory 40 stages stall and Write Back stage 50 keeps writing to the same register, if writing is enabled.

The stall mechanism is implemented in each stage by adding muxes in front of the registers 22 that pass the signals to the following stage. They allow either the next value or the actual value of the signal to go through the registers 22. When a stall situation is detected, a Stall Controller described below decides which stages of the pipeline has to be stalled and sets the corresponding select signals for the muxes.

If the select line of a mux indicates a stall, it passes the actual value of the signal to the register, otherwise the next value is passed and registered. This is shown in Fig. 4.

When an instruction is being fetched (Fetch stage 10) and the Instruction Memory is not ready, the whole pipeline is stalled. When Instruction Memory is ready again, the instruction is fetched, decoded, etc. as usual and the rest of the instructions already in the pipeline can also continue their operation.
The situation can be described as:

Clock cycle	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	IF	DE	EX	ME	WB	WB	WB	WB			
Instruction 2		IF	DE	EX	ME	ME	ME	ME	WB		
Instruction 3			IF	DE	EX	EX	EX	EX	ME	WB	
Instruction 4				IF	DE	DE	DE	DE	EX	ME	WB
Instruction 5					**IF**	IF	IF	**IF**	DE	EX	ME
Instruction 6									IF	DE	EX
Instruction 7										IF	DE

Instruction 5 cannot be fetched (clock cycle 5) because Instruction Memory is not ready, so each stage of the pipeline remains executing the same instruction until they are able to register out the results and receive new inputs. At clock cycle 8 instruction 5 can be fetched, so the normal processing continues.

When data memory is not ready, all the stages have to be stalled until data can be read from or written to memory again. The situation is described below:

Clock cycle	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	IF	DE	EX	ME	WB	WB	WB				
Instruction 2		IF	DE	EX	**ME**	ME	**ME**	WB			
Instruction 3			IF	DE	EX	EX	EX	ME	WB		
Instruction 4				IF	DE	DE	DE	EX	ME	WB	
Instruction 5					IF	IF	IF	DE	EX	ME	WB
Instruction 6								IF	DE	EX	ME
Instruction 7									IF	DE	EX

At clock cycle 5, data memory is not ready. From this point on, all the stages are stalled and instructions 1 to 5 cannot continue their execution until data memory is ready again. It occurs at clock cycle 7 and the stages finish the operation they were performing on clock cycle 5. Finally, in clock cycle 8 onwards the pipeline is working normally.

When the HALT instruction is decoded, the full pipeline is stalled. An interrupt will make the processor to continue the normal operation. The situation is illustrated below:

Clock cycle	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	IF	DE	EX	ME	WB	WB	WB	WB			
Instruction 2		IF	DE	EX	ME	ME	ME	ME	WB		
Instruction 3			IF	**DE**	EX	EX	EX	EX	ME	WB	
Instruction 4				IF	DE	DE	DE	**DE**	EX	ME	WB
Instruction 5					IF	IF	IF	IF	DE	EX	ME
Instruction 6									IF	DE	EX
Instruction 7										IF	DE

Instruction 4 is a HALT and after it is decoded (clock cycle 5 on) the full pipeline is stalled. In clock cycle 8 an interrupt is received which allows the pipeline to go on in the next clock cycle.

There are three classes of hazards, structural, data and control hazards. The processor of the invention has no structural hazards, due to all its units being fully pipelined. So, the hardware supports all possible combinations of instructions in simultaneous execution.

Data hazards are produced when the execution of instructions is dependent of

previous results. Control hazards arise when the instruction being executed tries to change the PC based on the result of instructions executed before but not yet finished.

The way hazards are detected is comparing the destination register of the three previous instructions with the source registers of the instruction currently being decoded. It allows the Decode stage 20 to take the appropriate actions in order to avoid mistaken situations. There is also, in some cases, a comparison to detect if the previous instructions are load instructions. In this case the actions to take are different, because the data appears in Memory stage 40 instead of Execution stage 30, as occurs when performing an ALU operation.

Data hazards can be classified according to the order of read and write accesses to registers in the program instructions. The hazard *write after write* is not present, because writing is only allowed during Write Back stage 50, so only one register is written at a time. The *write after read* hazard cannot happen because all Because all the reads are done in the Decode stage 20 which is earlier and writes are done in the write back stage 50 which later. Finally, the case *read after read* is not a hazard. Thus, the only data hazard present in EVE architecture is *read after write*.

Considering the following sequence of instructions executed in the pipelined processor:

		Clock cycle	1	2	3	4	5	6	7	8	9
1	ADD	R1,R2,R3	IF	DE	**EX**	ME	WB				
2	XOR	R4,R1,R5		IF	DE	**EX**	ME	WB			
3	SLL	R6,R7,R1			IF	DE	**EX**	ME	WB		
4	SRA	R8,R1,R9				IF	DE	**EX**	ME	WB	
5	SUB	R10,R1,R11					IF	DE	EX	ME	WB

The instructions 2, 3 and 4 try to read R1 before instruction 1 writes it back, so they have a *read after write* data hazard. This problem is managed by the use of forwarding, which means that the result of adding R2 and R3 (produced at the end of Execution stage 30 of instruction 1) is registered and bypassed as an input to the Execution stage 30 to be used by the second instruction. The same result is

registered out of Memory stage 40 in clock 4 and again it is forwarded to Execution stage 30 to be used by instruction 3. Finally, instruction 4 gets the contents of R1 by means of a forward from the Registers to the Execution stage 30 in clock cycle 6. Thus, instructions 2, 3 and 4 will have R1 in time and no stalls are required. Instruction 5 will not have problems because it will need R1 after it is written back (Write Back stage 50 of instruction 1).

The forwards are shown in Fig. 5. The data is permanently forwarded from the Execution and Memory stages 30 and 40 to the source operand muxes in the Execution stage 30. These values are only chosen if a hazard is detected in the Decode stage. In that case the appropriate select signals for the source operand muxes are set.

In the case of the forward from the Registers, it is not directly going to the source operand muxes but to the register muxes. Consequently, the value coming from the Registers is the forwarded value instead of the currently addressed register content.

There is one case of a *read after write* data hazard, which cannot be handled by forwarding. It is:

		Clock cycle	1	2	3	4	5	6	7
1	LW	R1,0(R2)	IF	DE	EX	**ME**	WB		
2	XOR	R3,R1,R4		IF	DE	**EX**	ME	WB	
3	AND	R5,R1,R6			IF	DE	EX	ME	WB
4	SLL	R7,R1,R8				IF	DE	EX	ME

The load instruction will not have the data until the end of the Memory stage 40 cycle and instruction 2 will need it at the beginning of Execution stage 30 cycle then, the pipeline has to stall for one clock. The new execution order can be represented as follows:

		Clock cycle	1	2	3	4	5	6	7	8
1	LW	R1,0(R2)	IF	DE	EX	**ME**	WB			
2	XOR	R3,R1,R4		IF	DE	DE	**EX**	ME	WB	
	NOP					EX	ME	WB		
3	AND	R5,R1,R6			IF	stall	DE	EX	ME	WB
4	SLL	R7,R1,R8					IF	DE	EX	ME

When this case occurs, the Fetch stage 10 is stalled, which means that PC is not changed and the same instruction (3) is fetched again. Because that, in clock 4 the inputs to the Decode stage 20 have not changed causing again the decoding of instruction 2.

The signals registered out by Decode stage 20 in clock 3 correspond to a NOP, which is equivalent to insert a delay to separate instructions 2 and 3. It guaranties that the rest of the pipeline continues working normally and the comparisons to detect hazards will have the correct sequence of instructions.

To know whether a branch instruction may or may not change the PC is necessary to wait until the comparison is made and it is also necessary to calculate the target address. In the processor those operations are done during Decode stage 20, so no stalls are required from that point of view. However, the register used in the comparison could be still in use by previous instructions, incurring in a control hazard. The situation is similar in JR and JALR instructions, since they require the contents of register in Decode stage 20 to become the target address.

Some of those control hazards can be avoided by forwarding, similar to the data hazards, but there are three situations that require a stall. In any case, the instruction following a taken branch, JALR or JR will be annulled. If the branch is not taken, the next instruction will be normally executed.

The next example illustrates the situation where a Branch taken (as well as JR or JALR) annuls next instruction and an untaken Branch allows it to continue. In that case, there are no control hazards so no forwards or stalls are required.

Untaken branch	IF	**DE**	EX	ME	WB			
Instruction i+1		IF	DE	EX	ME	WB		
Instruction i+2			IF	DE	EX	ME	WB	
Instruction i+3				IF	DE	EX	ME	WB

Taken branch	IF	**DE**	EX	ME	WB			
Instruction i+1		IF	-	-	-	-		
Branch target			IF	DE	EX	ME	WB	
Branch target +1				IF	DE	EX	ME	WB

This situation below causes control hazards which require forwarding:

		Clock cycle	1	2	3	4	5	6	7	8	9
1	SUB	R1,R2,R3	IF	DE	**EX**	**ME**	**WB**				
2	ADD	R4,R4,R1		IF	DE	EX	ME	WB			
3	BEQ	R1,imm1			IF	**DE**	EX	ME	WB		
4	BNE	R1,imm2				IF	**DE**	EX	ME	WB	
5	JR	R1,imm3					IF	**DE**	EX	ME	WB

Instruction 1 calculates R1, which is ready at the end of clock cycle 3. Instruction 2 needs R1 at the beginning of clock cycle 4, but that situation is fixed by forwarding, as explained when discussion data hazards. Instructions 3, 4 and 5 will need R1 in Decode stage 20 because it is used to decide whether the branch is taken or not or to calculate next PC in the case of the JR.

The required forwards come from Execution stage 30, Memory stage 40 and the registers in the edges of clock cycles 4, 5 and 6 respectively, to Decode stage 20 at clock cycles 4, 5 and 6. By means of these forwards, R1 will be present in time and it will not incur in stalls.

The worst cases arise when there are control hazards that need to stall the pipeline until they get the value they were waiting for. There are three cases. The first occurs when the source register of the instruction being decoded is the destination register of the previous instruction:

		Clock cycle	1	2	3	4	5	6	7
1	SUB	R1,R2,R3	IF	DE	**EX**	ME	WB		
2	BEQ	R1,imm1		IF	**DE**	EX	ME	WB	
3	XOR	R4,R5,R6			IF	DE	EX	ME	WB

In that case, R1 is needed at the beginning of clock cycle 3 by instruction 2 but produced at the end of that clock cycle by instruction 1. This situation requires one clock stall as follows:

		Clock cycle	1	2	3	4	5	6	7	8
1	SUB	R1,R2,R3	IF	DE	**EX**	ME	WB			
2	BEQ	R1,imm1		IF	DE	**DE**	EX	ME	WB	
	NOP					EX	ME	WB		
3	XOR	R4,R5,R6			IF	stall	DE	EX	ME	WB

In clock cycle 3 the situation is detected by means of the comparison between the source operands (only one in this case) of the instruction being decoded and the destination register of the three previous instructions. It yields in stalling Fetch stage 10 and sending a NOP through the pipe from Execution stage 30 on. One clock later, R1 has been produced and is available to Decode stage 20 by means of a forward from Execution stage.

The second case is similar to the one just explained, but the instruction causing the hazard is now a load:

		Clock cycle	1	2	3	4	5	6	7
1	LW	R1,R2,R3	IF	DE	**EX**	ME	WB		
2	JR	R1,imm1		IF	**DE**	EX	ME	WB	
3	AND	R4,R5,R6			IF	DE	EX	ME	WB

Instruction 1 will produce R1 in Memory stage 40, but it is required by instruction 2 in Decode stage, which leads in a two-clock stall. The execution will be in this order:

		Clock cycle	1	2	3	4	5	6	7	8	9
1	LW	R1,R2,R3	IF	DE	**EX**	ME	WB				
2	JR	R1,imm1		IF	DE	DE	**DE**	EX	ME	WB	
	NOP					EX	ME	WB			
	NOP						EX	ME	WB		
3	AND	R4,R5,R6			IF	stall	stall	DE	EX	ME	WB

Here, the fetch stage 10 has to stall twice to allow instruction 1 to produce the data

required by instruction 2. In clock 5 a forward will bring R1 from Memory stage 40 to Decode stage 20 where is required to calculate the target address for the jump. Meanwhile two NOP instructions are sent through the pipeline to keep it working normally.

The third control hazard causing a stall is described as follows:

		Clock cycle	1	2	3	4	5	6	7	8
1	LW	R1,R2,R3	IF	DE	**EX**	**ME**	WB			
2	SUB	R7,R8,R9		IF	**DE**	**EX**	ME	WB		
3	JR	R1,imm1			IF	**DE**	**EX**	ME	WB	
4	AND	R4,R5,R6				IF	**DE**	**EX**	ME	WB

R1 is produced and required in the same clock cycle, so the situation needs one-clock stall to be executed properly. Considering the stall, the execution order is now as showed below:

		Clock cycle	1	2	3	4	5	6	7	8	9
1	LW	R1,R2,R3	IF	DE	EX	**ME**	WB				
2	SUB	R7,R8,R9		IF	DE	EX	ME	WB			
3	JR	R1,imm1			IF	DE	**DE**	EX	ME	WB	
	NOP						EX	ME	WB		
4	AND	R4,R5,R6				IF	stall	DE	EX	ME	WB

More detail with the addition of the extra hardware to prevent stalls and hazards, it is the muxes in front of the registers and the forwards, the pipeline shown in Fig. 3 becomes the one shown in Fig. 6.

The elements present in each stage will be explained in more detail below.

The following Tables give a complete dictionary of the various signals used. Tables 3 to 12 inclusive below show various top level signals, while Table 13 shows the instruction unit internal signals. Table 14 shows the execution unit internal signals and Table 15, the data unit internal signals.

Table 3 – Signals between Instruction Unit and Register Sets

AddrReg1[5:0]	This is the address of the register that is being used as the first operand of the instruction being decoded.
AddrReg2[5:0]	This is the address of the register that is being used as the second operand of the instruction being decoded.
EveInt	When an external interrupt has been signaled, the interrupt logic asserts this signal to inform the processor that an interrupt has occurred and it must jump to the interrupt service routine to service this interrupt.
IntAck	After the processor has seen the interrupt flag; *eveInt*, and starts to service the interrupt, it asserts this acknowledge signal so that the interrupt logic knows to acknowledge the external interrupt source.
RegS1[n-1:0]	This is the data from the General Purpose Register selected by *addrReg1[5:0]* before it is pipelined through to the Execution stage. This signal connects to the instruction unit where the register value is to be used in the generation of the next instruction address.
RegS2[n-1:0]	This is the data from the General Purpose Register selected by *addrReg2[5:0]* before it is pipelined through to the Execution stage. This signal connects to the instruction unit where the register value is to be used in the generation of the next instruction address.
RegS1Sel[1:0]	Both a GPR and a SR has been selected, but it may also be a case of a stall or a forward, so before the data is pipelined through as operand 1 this select line selects one of the four sources.
RegS2Sel[1:0]	Both a GPR and a SR has been selected, but it may also be a case of a stall or a forward, so before the data is pipelined through as operand 2 this select line selects one of the four sources.
SReset	If the processor receives an illegal instruction this signal is set HIGH for one clock cycle to set the Reason Register correctly and to clear pending interrupts.

Table 4 - Signals between Instruction Unit and Instruction Memory

imAddr[m-1:0]	This is the instruction memory address bus. It is a registered output. It is in byte address sizes but all values that appear on it are word addresses. On a reset this bus goes to zero.
imData[p-1:0]	This is the data from the instruction memory i.e. it is the instruction addressed by the instruction memory address bus.
instRdy	This signal indicates when valid data is available from the instruction memory. If the instruction memory takes more than a clock cycle to produce valid data from when it is addressed, this signal must be pulled low until valid data is available.

Table 5 - Signals between Instruction Unit and data memory

dataRdy	This input signal indicates when valid data is available from the data memory. If the data m mory takes more than a clock cycle to produce valid data from wh n it is addressed, this signal must be pulled low until valid data is available.

Table 6 - Signals between Instruction Unit and Data Unit

memStgStall	This signal is asserted **HIGH** by the stalling logic if the Memory stage of the processor has to be stalled.

Table 7 - Signals between Instruction Unit and Execution Unit

aluOp[4:0]	This signal is the function code of the operation to be performed by the ALU on the two selected source operands.
aluRes[n-1:0]	
carry	This signal is the carry bit stored. It is passed from the Execution Unit to the Register Unit to be used for the Branch on Carry instructions in case that they are being implemented.
exDestReg[n-1:0]	
exLoad	Signal indicating that a Load instruction is in the execution stage. This is used in the detection of Hazard conditions.
exStgStall	This signal is asserted **HIGH** by the stalling logic if the Execute stage of the processor has to be stalled.
instDestReg[5:0]	This is the address of the register that the result of the instruction operation is destined for. This is passed through the pipeline in parallel with the instruction executing. This signal is when this destination register address is being passed through the Execution stage.
instDMCS	This signal partners *instDataMemCtrl[3:0]* in that it is a select signal that tells the data memory Controller in the Memory stage that an access to memory is happening. This is the signal in the Execute stage of the pipeline.
instDMCtrl[3:0]	This signal indicates to the data memory Controller in the Memory stage which type of operation is performed in order to assert the appropriate control signals of the data memory. This is the signal in the Execute stage of the pipeline.
instImm[n-1:0]	The Decode stage determines the Immediate data of an instruction and passes it through sign extension logic. This signal is the registered 32-bit Immediate data into the Execution stage.
instPC[n-1:0]	This is the Program Counter registered out of the Decode stage into the Execution stage of the pipeline.
instWrRegEn	If the instruction being executed results in a write to a register, this signal is asserted. Similar to *instDestReg[5:0]*, this signal is passed through the pipeline in parallel with the instruction executing. Again, this is the signal during the Execution stage of the instruction.
resValid	
s1MuxSel[2:0]	The first operand of the ALU is selected from 6 sources including forwarded data. For the instruction in the Execution stage, this signal selects the source for this operand.
s2MuxSel[2:0]	Similar to *s1MuxSel[2:0]*, this signal selects the source for the second operand of the ALU.
s2PassMuxSel[1:0]	If the operation being performed by the instruction results in writing to a register, the correct data source of either the ALU (a Register-to-Register function) or the data memory (a Load operation) is selected as the data sourc for the regist r write. Again this is similar to *instDestReg[5:0]* in the Execution stage.
useCarry	This signal informs the ALU that the instruction being executed must use the carry bit stored.

Table 8 - Signals between Execution Unit and Data Unit

aluRes[n-1:0]	This is the result of the operation performed by the ALU and has been registered into the Memory stage of the pipeline. It contains either a value to be written to a register or an address of a memory location. This bus is passed back to the Instruction Unit so when there is a data hazard case where the data on this bus is required in the Decode stage of an instruction cycle it is available.
dmCtrl[3:0]	This is the memory control signal in the Memory stage. It is the control signal for the memory control logic. When the data memory chip select signal is high, the data on this bus is valid. 0 00 0 – Store Byte 0 01 0 – Store Halfword 0 10 0 – Store Word 1 00 0 – Load Byte Unsigned 1 00 1 – Load Byte Signed 1 01 0 – Load Halfword Unsigned 1 01 1 – Load Halfword Signed 1 10 0 – Load Word
dmCS	When this signal is HIGH it indicates that an access to memory is occurring. This data memory chip select is the same as the external signal.
exDestReg[5:0]	This is the Destination register address passing through the pipeline, parallel to the instruction, in the Memory stage of the pipeline. This is also passed back to the Instruction unit where it is used in the detection of data hazards.
exWrRegEn	This is the write to Destination register signal passing through the pipeline, parallel to the instruction, in the Memory stage of the pipeline.

Table 9 - Signals between Execution Unit and data memory

dmDataOut[n-1:0]	If a Store to memory instruction occurs, the data to be written to the data memory location, addressed by *dmAddr[n-1:0]*, appears on this bus. This is a registered output.
dmAddr[q-1:0]	During accesses to data memory, the address of the data location appears on this bus. It is a registered output. The addresses that appear on this bus are byte addresses.
DmCS	When this signal is HIGH it indicates that an access to memory is occurring.

Table 10 Signals between Data Unit and General Purpose and Special Registers

addrDestReg[5:0]	This is the address of the Destination register in the Write Back stage of the pipeline where it is used to address the register in the register file that the data on bus *dataBack[31:0]* is destined for. This signal is also passed back into the Instruction Unit where it is used in the detection of data hazards.
dataBack[n-1:0]	When an instruction comes through to the final Write Back stage of the pipeline, any data that was calculated or read from memory will appear on this bus to be written to a register in the register files. This signal is forwarded back to both the Instruction Unit and Execution Unit where the data may be required to cover for data hazards.
wrRegEn	If the instruction in the Write Back stage of the pipeline is a valid write to a register this signal will have been asserted when the instruction was in the Decode stage and will have propagated through the pipeline and is used here in the Write Back stage.

Table 11 - Signals between data memory and Data Unit

dmDataIn[n-1:0]	If a Load from memory instruction occurs, the data from the data memory location, addressed by *dmAddr[31:0]*, is passed to the processor on this bus.
dmRW	This signal indicates to the memory whether a load or store is happening. If it is HIGH this indicates a store to memory and if it is LOW a load from memory is happening.
dmSize[1:0]	This output signal is used by the processor to indicate to the data memory when word, halfword or byte transfers are required. **b00** indicates that the transfer is a byte, **b01** indicates that the transfer is a halfword and **b10** indicates that the transfer is a word. These values are valid for both loads and stores.

Table 12 – General Signals

sysClk	This is the system wide clock. It is the only clock domain in the EVE processor so all signals are relative to it.
sysReset	This is the system wide RESET signal. It is synchronous to *sysClk*.

Table 13 - Instruction Unit Internal signals

AluOperation[4:0]	This is the decoded ALU operation before it is registered into the Execution stage.
BitSelect	

brMux1Byte0Sel[1:0]	This signal selects the appropriate byte of the correct source of data for the comparison of the highest byte of data being compared. 00 – regS1 01 – aluRes 10 – dataBack 11 – zero
brMux1Byte1Sel[1:0]	This signal selects the appropriate byte of the correct source of data for the comparison of the second highest byte of data being compared. 00 – regS1 01 – aluRes 10 – dataBack 11 – zero
brMux1Byte2Sel[1:0]	This signal selects the appropriate byte of the correct source of data for the comparison of the second lowest byte of data being compared. 00 – regS1 01 – aluRes 10 – dataBack 11 – zero
brMux1Byte3Sel[1:0]	This signal selects the appropriate byte of the correct source of data for the comparison of the lowest byte of data being compared. 00 – regS1 01 – aluRes 10 – dataBack 11 – zero
brMux2Byte0Sel	This signal selects the correct data item that the highest byte of data being is being compared with.
brMux2Byte1Sel	This signal selects the correct data item that the second highest byte of data being is being compared with.
brMux2Byte2Sel	This signal selects the correct data item that the second lowest byte of data being is being compared with.
brMux2Byte3Sel	This signal selects the correct data item that the lowest byte of data being is being compared with.
ZeroRes	This signal is asserted **HIGH** by the comparator if the two values being compared are equal otherwise it is a **LOW**.
DecStgStall	This signal is asserted **HIGH** by the stalling logic if the Decode stage of the processor has to be stalled.
destReg[4:0]	This is the address of the destination register before it is registered into the Execution stage.
DMemCS	This signal indicates to the memory controller that a memory access is occurring. This is the signal decoded, before it is registered into the Execution stage.
dMemCtrl[3:0]	This signal indicates to the data memory controller what type of memory access is to be performed by this instruction in order for it to assert the appropriate control signals of the data memory. This is the signal decoded, before it is registered into the Execution stage.
FetchStgStall	This signal is asserted **HIGH** by the stalling logic if the Fetch stage of the processor has to be stalled.
HaltInst	When a Halt instruction is decoded, the processor must remain in the same state until it is released. This signal tells the stall logic to stall every stage of the pip lin .

HazardDelay	This signal indicates to the stalling logic that a data or a control hazard has been detected and it can only be avoided by letting a previous instruction complete either the Execution or Memory stages and stalling the Fetch stage for a clock cycle.
immediate[25:0]	This is the immediate part of the instruction registered into the Decode stage of the pipeline. Up to 26 bits of the instruction are used for the immediate, depending on the instruction type, so it is up to the sign extension logic to determine how much of this bus to use when generating sign extended data.
ImmExtended[n-1:0]	This is the immediate data extended to the machine word length.
ImmOffset	
PC[31:0]	This is the program counter. It is the same bus as *imAddr[m-1:0]*.
PCsel[2:0]	This is the control signal for the selection of the next value for the program counter. 000 – selects PC + 4 as next address 001 – selects address 0 as next address (Reset) 010 – selects the present PC value as the next address (Stall) 011 – selects address 4 as next address (Interrupt/Trap) 100 – selects PC + *immExtended[m-1:0]* as the branch or jump address 101 – selects *regS1Data[m-1:0]* as next address (Jump to GP Register value) 110 – selects *pcReturnAddr[m-1:0]* as next address (Jump to Special Register value)
RegisteredInst[p-1:0]	This is the next instruction registered from the Fetch stage of the pipeline.
s1Sel[2:0]	This is the S1 Mux Select signal before it is registered into the Execution stage.
s2PassSel[1:0]	This is the signal that selects the data for memory in the execution stage before it is registered into the Execution stage.
s2Sel[2:0]	This is the S2 Mux Select signal before it is registered into the Execution stage.
SelSR1	
SelSR2	
SignExtendSel[1:0]	This is the control signal that tells the sign extension logic how the immediate data should be extended. 00 – *immExtended[31:16]* = 0, *immExtended[15:0] = immediate[15:0]* 01 – *immExtended[31:16]* = each bit set to value of *immediate[15]*, *immExtended[15:0] = immediate[15:0]* 1x – *immExtended[31:26]* = 0, *immExtended[25:0] = immediate[25:0]*
TakestageDataS1	
TakestageDataS2	
TestBit	
UsePrevCarry	
WriteRegEn	This is the write enable signal for the destination register before it is registered into the Execution stage.

Table 14 - Execution Unit Internal signals

aluS1[n-1:0]	This is the first operand for the ALU selected from a register source, the program counter and two forwarded data.
aluS2[n-1:0]	This is the second operand for the ALU selected from a register source, an immediate data obtained from the instruction and two forwarded data.
RepData[n-1:0]	On a write to memory, the data that is being written to memory is set up in the Ex cution stage of the pipeline. This bus contains the selected data source from a register selected as the second operand or one of two forwarded data.

StData [n-1:0]	When performing a store byte or halfword, the data to be stored is replicated up through the word in order to have the byte or halfword of data in the correct location in the word for storing to memory. This is the data to be stored before it is pipelined through to the Memory stage.

Table 15 - Data Unit Internal signals

AdjDMDataIn[n-1:0]	This is the data read from memory that has been adjusted accordingly in the data memory sign extension unit.
LoadSel	This signal controls the write back mux to chose from: 0 – aluRes[n-1:0] 1 – adjDMDataIn[n-1:0]
DmSESel[2:0]	This is the control signal that tells the data memory sign extension unit how it has to extend the data being read from memory. 1xx – *dmDataIn [n-1:0]* is allowed to pass through 011 - Sign Extend byte 001 - Sign Extend halfword 010 - Zero Extend byte 000 - Zero Extend halfword

Fig. 7 illustrates the top level break down of the core into 4 main areas.

There is shown four essential areas namely the Instruction Unit 61, the register unit 62, the execution unit 63 and the data unit 64.

The instruction unit 61 is the section where all work is done with the instruction, control of fetching it from the instruction memory, decoding it and setting control signals for the rest of the processors.

The register unit 62 is separated from the Instruction Unit. This is aimed towards synthesis, as there will be a large amount of actual registers implemented. It is addressed by the decoding of the instruction to present operands to the Execution Unit.

The execution unit 63 is the implementation of the Execution stage of the EVE Processor pipeline. This is in its own block as concentration can be put on it because of its importance and possibly its critical timing.

Finally, the Data Unit 64 is the remainder of the processor, which in effect does

something with data, writes data to memory reads data from memory and then writes data back to a register of the processor.

Each of these are dealt with in detail below.

The instruction unit 61 effectively comprises the Fetch stage and the Decode stage of the processor. They are grouped under the name of Instruction Unit. The block is subdivided into nine sub-blocks: Sequencer, Fetch stage Registers, Sign Extender, Forwarding Decision Logic, Comparator, Source Selectors, Decoder, Decode stage Registers and Stall Controller.

The Sequencer and the Fetch stage Registers form the Fetch stage of the pipeline. The other seven sub-blocks form the Decode stage. Included here is the Stall Controller, which is a combinational block that generates the control signals to **stall** the whole pipeline operation when required. The Information Flow Diagram with the signals interconnecting the sub-blocks is shown in Fig. 8.

The main sub-block in the Fetch stage is the Sequencer. It has two adders, a mux and a register. Its block diagram is shown in Fig. 9. The mux passes the value of next PC depending on the select signal generated by the Decoder and this value is registered out as new PC. The PC mux has 9 inputs:

- PC+4, which is used when the normal flow of the program is followed. It is also the default option.
- 0x0, used when a reset signal is received or an invalid instruction is detected.
- 0x4, the address from where the Exception Service Routine starts.
- PC+imm, it is the target address for branch, J and JAL instructions.
- PC, this value is forwarded back to the mux to be chosen when the case of a stall of the Fetch stage happens.
- *regS1*, it is the output of the GPR addressed by JR or JALR instructions.
- *pcReturnAddr*, used when loading LAR, after servicing a routine called by JAL or JALR instructions, or EAR, after completing the servicing of the Exception Service Routine.

- *aluRes*, this value is forwarded from the Execution stage to be used when there is a control hazard.
- *dataBack*, this value is forwarded from the Memory stage to be used when there is a control hazard

The decision of choosing next PC is based on the instruction being decoded and the presence of control hazards. The encoding values for PCSel are listed in Table 16.

Table 16

Code	Operand	Label	Comments
0000	0x0	RESETCPU	Reset
0001	PC + 4	CONTINUE	Continue
0010	PC	STALLCPU	Stall
0011	PC + *imm*	BRJMP	Branch/Jump
0100	0x4	EXCEPTION	Trap/Interrupt
0101	regS1	JMPREG	Jump to Register value
0110	pcReturnAddr	RETURN	Return to Link address
1000	aluRes	PCEXDATA	Forwarded data from Execution stage
1001	dataBack	PCMEMDATA	Forwarded data from Memory stage
1010	unused		
1011	Unused		
1100	Unused		
1101	Unused		
1110	Unused		
1111	Unused		

The Fetch stage Register sub-block only comprises a mux and a register. It takes the data coming from the Instruction Memory and registers it into the following stage. In the case of a Fetch stage stall (*fetchStgStall* asserted), the mux chooses the previous instruction registered instead of the actual until the stall condition is over.

This is the most complex stage of the processor due all the control decisions are taken there. The six sub-blocks that form this stage work in order to decode the

instruction coming from the previous stage and generate the control signals that will be sent through the pipeline. Those signals depend not only on the instruction itself, but also on other events like data and control hazards, stalls and interrupts.

The Sign Extender receives the corresponding bits of the instruction from the Fetch stage. The way they are converted into a machine width word is determined by the signal *signExtendSel* generated by the Decoder. There are four possibilities:

- The 16 bottom bits are sign extended. This is chosen when the resultant word will be used in a signed ALU operation.
- The 16 bottom bits are padded with zeros. In this case the ALU operation is unsigned.
- The 26-bit input is sign extended or truncated to the width of the instruction memory address bus and when it is added to PC will generate the target address of a branch, J or JAL instruction.
- The fourth option is swapping the bottom machine halfword bits of the immediate data into the top machine halfword bits of the resultant word and clearing its bottom machine halfword bits. This implementation is used when an LHI instruction is decoded.

This sub-block compares the register address of the source operands of the instruction being decoded with the destination register address of the three previous instructions. The comparison detects if there are hazards present in the instruction flow. If so, this block indicates which forwards should be taken in order to avoid the hazard condition. The flow diagram of the sub-block is shown in Fig. 10.

The hazards can be present for any source register, so the flow diagram shown in the figure is repeated twice in order to cover the two source operands. The one comparing the first source operand produce a signal called *takestageDataS1* that indicates which forward should be chosen in order to avoid the hazard. For the comparison of the second source operand the signal produced is called *takestageDataS2*.

The encoding values for *takestageDataS1* and *takestageDataS2* are the following:

Table 17

Code	Label	Comments
00	TakeRegData	Use content of the register addressed by the instruction
01	TakeExData	Use *aluRes*
10	TakeMemData	Use *dataBack*
11	TakeWbData	Use *dataBack* after bypassing GRP and SR

The Comparator is used to decide whether a branch has to be taken or not. Fig. 11 shows a block diagram of the implementation of this branch decision logic. It comprises of a set of 8 bit wide muxes, sets of 8 bit wide XOR gates and a comparator. There are two types of muxes used for each 8-bit element of the comparator. The first chooses between the source register addressed by the branch instruction or one of the two forwards (*aluRes* and *dataBack*) in case there are control hazards present. Zero also feeds into this mux to cover the case where this byte is irrelevant to the comparison but it cannot affect the result. The second mux chooses the data that the register value is to be compared against. For the Branch if Byte equal to Immediate instructions, the 8 bit immediate from the instruction is passed through. If it is a compare of register values the corresponding byte of the register selected by Rs2 in the instruction is passed through otherwise, zero is passed through to be compared against the value. A bitwise XOR is then performed between each pair of outputs from the muxes, producing an 8-bit number. This is combined with others to form a data item of machine data width, which is then compared with zero. A 1-bit result *zeroRes* is then generated to inform other blocks of logic that the compare was True or False.

The comparison unit can be fully parameterised, allowing any data path size comparisons. The sub-block of two muxes and XOR gates are parameterised to accept data from 1 up to 8 bits. Depending then on the data path size, any number of these sub-blocks can be instantiated to form the data path width. For example, a 20-bit data path processor will require 2 8-bit and 1 4-bit sub-blocks. The final block, which compares with zero, will be of data path size.

The restriction of this design implementation is that halfword comparisons can only be made on multiple of 8 data path width configurations. Of course byte comparisons are only allowed when the data path size is bigger that or equal to 8. When performing a byte comparison, the rest of the sub-blocks will force the output of the XOR gate to be zero by comparing zero with zero. This way, the bits that are not being tested will not affect the final comparison.

The decoder sub-block is where the main decisions of the processor are taken. It receives the instruction to be decoded from the Fetch stage and additional information from other sub-blocks in order to generate the control signals for the whole pipeline. The flow diagram of the Decoder is shown in Figs 12 and 13.

The first step is checking the signal *annul*. If asserted, the instruction contents are cleared because it must not be executed. In this case a NOP (ADD R0, R0, R0) will be sent to the pipeline and the PC value corresponding to that instruction will be stored in the proper register, depending on the cause of the annul.
Then the addresses (*addrReg1*, *addrReg2*) are assigned directly from the instruction. Although, there is one exception concerning to the position of the addresses in the instruction. The address for the second source operand appears in a different place for store instructions, so it has to be picked from the right place. Thus, the indicators and addresses are assigned as mentioned above, but they will be defaulted (to R0 in the case of the register addresses) when decoding each specific instruction if they are not addressing any register or they only can address GPR.

The *eveInt* signal is the next to be checked. When asserted, two more checks have to be done before decide whether to service the interrupt or not. The first examines the instruction being decoded. If it is an instruction that will change the value of PC in the next clock cycle and thus, will set signal *annul,* like jumps, TRAP, RFE, RET or branches, the interrupt service has to be delayed in order to avoid storing the wrong value of PC. The second looks for a TRAP or another interrupt currently being serviced (signal *inESR* set). If so, the interrupt will be serviced when the Exception Service Routine finishes because both, TRAP and interrupt store PC in EAR and it would be overwritten.

If the service routine of the interrupt can start in the actual clock cycle, the following control signals are asserted:

- The signal *annul* to avoid the execution of the instruction being fetched
- The value of PC is stored in EAR, so the signal *writeEAR* is set.
- The signal *intESR* is asserted in order to prevent a TRAP or another interrupt to be serviced while servicing it.
- The next value of PC has to be 0x4, so the appropriate value of *PCsel* is chosen.
- The signal *intAck* is asserted to acknowledge the interrupt.

When there is not an interrupt or it cannot be serviced from the present clock cycle on, the flow of the decoding continues by checking the type of instruction received. Depending on the instruction and the possible hazards detected, the corresponding control signals are set or defaulted.

Data hazards can affect the execution of R-type or I-type (except branches) instructions. Thus, the signal *s1Sel* is asserted accordingly to *takestageDataS1* and the signals *s2Sel* and *s2PassSel* accordingly to *takestageDataS2* in order to chose the source operands in the Execution Unit. They will select either the default data if there are no hazards present or the forward indicated by the corresponding *takestageData* signal.

Control hazards can affect JR, JALR and branch instructions. These instructions only have one source operand, so only *takestageDataS1* is checked in order to select the proper data. In the case of a branch, the signal *compSel* is set to allow either the data or a forward to be compared with zero. For JR or JALR, *PCsel* is set in order to choose either the register addressed by the instruction or a forward to become the target address.

In both cases of hazards the specific decode logic of each instruction checks for stalls. These checks include not only the *takestageData* signals but also the signals *instLoad* and *exLoad*. They indicate if the two previous instructions were loads. If a

hazard causes a stall, the signal *hazardDelay* is sent to the Stall Controller where the appropriate actions are taken. A NOP is also sent to the pipeline as explained.

When decoding a HALT instruction, the signal *haltInst* is set. It indicates to the Stall Controller that the whole pipeline has to be stalled until receiving an interrupt. The signal *aluOperation* is set depending on the field AluOp of R-type and I-type instructions. When defaulted it indicates a signed addition. When the destination register has to be written, the signal *writeRegEn* is set as well as *destReg* address. They are sent through the pipeline and will be effective in the Write Back stage of the corresponding instructions.

The *signExtendSel* signal indicates to the Sign Extender which type of extension has to perform on the data. Even though, the sign extended value is only used by I-type, J, JAL and LHI instructions.

When decoding load and store instructions, *dMemCtrl* and dMemCS are accordingly asserted. They are sent through the pipeline to indicate the data memory Controller (in the Memory stage) which signals have to assert. It contains information about the instruction to perform (load or store), the size of the access (byte, halfword or word) and the sign extension that has to be done on the data when loading from data memory.

The summary of the control signals generated by the Decoder is shown in Table 18. Signals 1 to 4 are directly assigned from the instruction and later on defaulted if required. Signals 5 to 12 go to the Decode stage Registers sub-block to be registered before being sent to the Execution stage. Signals 13 to 18 are directly connected to other sub-blocks of the Instruction Unit. Finally, signals 19 and 20 are internal to the Decoder.

Control Signal	R-R	Alu/Imm	Load	Store	LHI	JAL	JALR	J	JR	Branch	RET	TRAP	RFE	HALT
AddrReg1	set	set	set	set	set	Set	set	set	set	set	set	set	set	set
AddrReg2	set	set	set	set	set	Set	set	set	set	set	set	set	set	set
SelSR1	set	default	default	default	default	Default	default	default	default	default	set	default	default	default
SelSR2	set	default	default	default	default	Default	default	default	default	default	default	default	default	default
DestReg	set	set	set	default	set	Default	default	default	default	default	default	default	default	default
AluOperation	set	set	default	default	default	Default	default	default	default	default	default	default	default	default
s1Sel*	set	set	set	set	set	Set	set	set	set	set	set	set	set	set
s2Sel*	set	set	set	set	set	Set	set	set	set	set	set	set	set	set
s2PassSel*	set	set	set	set	set	Set	set	set	set	set	set	set	set	set
WriteRegEn	set	set	set	default	set	Default	default	default	default	default	default	default	default	default
DmCtrl	default	default	set	set	default	Default	default	default	default	default	default	default	default	default
SignExtSel	default	se/pz 16b	se 16b	se 16b	swap	se 26b	default	se 26b	default	se 16b	default	default	default	default
DmCS	default	default	set	set	default	Default	default	default	default	default	default	default	default	default
PCsel*	PC+4	PC+4	PC+4	PC+4	PC+4	PC+off	regS1*	PC+off	regS1*	if taken PC+off else PC+4	LAR	0x4	EAR	PC
Annul	default	default	default	default	default	Set	set	set	set	set if taken	set	set	set	default
WriteLAR	default	default	default	default	default	Set	set	default	default	default	default	default	default	default
WriteEAR	default	default	default	default	default	Default	default	default	default	default	default	set	default	default
HaltInst	default	default	default	default	default	Default	default	default	default	default	default	default	default	set
HazardDelay	determine	determine	determine	determine	default	Default	determine	default	determine	determine	default	default	default	default
TrapESR	default	default	default	default	default	Default	default	Default	default	default	default	set	default	default
IntESR	default	default	default	default	default	Default	default	Default	default	default	default	default	default	default
SReset	determine	determine	determine	determine	determine	Determine	determine	Determine	determine	determine	determine	determine	determine	determine
IntAck	default	default	default	default	default	Default	default	Default	default	default	default	default	default	default

* Hazard dependent

Table 18- Summary of the control signals generated by the Decoder

The flow charts corresponding to the different types of instructions are shown in Figs. 14 to 33. They illustrate the actions taken when decoding every specific instruction.

The register selectors block generates two control signals for the General Purpose and Special Registers block. The signals are *reg1Sel* and *reg2Sel*. They chose the adequate registers content that are sent to the Execution stage. They are identically generated, so only *reg1Sel* will be explained here.

To set the signal, the sub-block receives information about the stalls affecting the Decode stage (*decStgStall*), the possible hazards detected (*takestageDataS1*) and which bank of registers is being addressed in each case, the GPR or the SR (*selSR1*). Accordingly, *reg1Sel* is generated as shown in Fig. 33.

During a Decode stage stall, the value selected is the one previously registered. In the case of a data hazard requiring forwarding from the Write Back stage, the signal chosen is *dataBack*. Finally, in a normal situation, the data selected is the content of the register addressed by *addrReg1* in the group of registers indicated by *selSR1* (when the signal is asserted, the SR is selected, otherwise is the GPR).

The encoding values for *reg1Sel* are listed in Table 19:

Table 19 - Encoding values for the signal *reg1Sel*.

reg1Sel/reg2Sel	Label	Comments
00	STALLDATA	Select previous value
01	DATABACK	Select *dataBack* value
10	SREGDATA	Select the content of the SR addressed
11	GPREGDATA	Select the content of the GPR addressed

The decode stage registers transmit the data and control signals produced in the Decode stage to the Execution stage. They have a mux in front of them and there are two values to choose from, the actual value, which is selected under normal

conditions, and the previous value, that is selected during a Decode stage stall. The signal that controls these muxes is *decodeStgStall*. When it is asserted the value selected is the previous value, otherwise it selects the one currently produced.

The signals muxed and registered to the Execution Unit are: *aluOp, s1MuxSel, s2Muxsel, s2PassMuxSel, instWrRegEn, instDMCS, instDMCtrl, instDestReg, instImm* and *instPC*. The prefix inst is added to the signal names to indicate that the corresponding signals have been registered out the Decode stage.

The stall controller is a combinational block which receives information from data memory, Instruction Memory and the Decoder about the situations that require the pipeline to stop. The flow diagram for this sub-block is illustrated in Fig. 34.

There are three situations that require the whole pipeline to stop. They are Data or Instruction Memories not ready and HALT instruction. In the case of the memories, the signals *dataRdy* and *instRdy* respectively are asserted in order to indicate to the Stall Controller that has to stall. To do so, the signals *decStgStall*, *exStgStall*, *memStgStall* and *fetchStgStall* are asserted preventing the corresponding stages to send new signals through the pipeline. When *dataRdy* or *instRdy* are deasserted, so are the stall signals and the normal operation is allowed to continue.

When decoding a HALT instruction, the Decoder asserts the signal *haltInst* to the Stall Controller. It has the same effect as *dataRdy* or *instRdy*, so the whole pipeline is stalled. In this case, receiving an interrupt will restart the normal operation.

When a data or a control hazard causes a stall, the signal *hazardDelay* is asserted by the Decoder. It makes the Fetch stage to stall for one clock cycle. If the hazards persists after the stall, *hazardDelay* will be asserted again causing the Fetch stage to stall once more.

The Execution Unit 63 is at the heart of the processor. Although, all the decoding is done in the Instruction Unit 61, it is in the Execution unit 63 that most instructions are implemented. This unit completely encompasses the Execution stage of the processor pipeline from when the data and control signals enter the execution stage until they are registered out into the next pipeline stage, the Memory stage.

Shown in Fig. 35, is a block diagram of the execution stage. The Arithmetic and Logic Unit (ALU) is where all functions are performed on two selected operands. For register to register instructions and the jump and link instructions a value is generated in the ALU and this is written back to a destination register. For data memory Load and Store instructions, the value generated in the ALU is the effective memory address.

Specifically for Memory Store instructions, the data to be stored in memory is passed through a different route through the Execution Unit. This is through the Source Operand Muxes and then through a replication unit, which is described below. The SourceOperandMuxes is where the operand selection occurs and finally the Execution stage Registers controls the registering and stalling into the next pipeline stage, the Memory stage.

Some of the operations done by the ALU set a carry flag, which is sent to the Instruction Unit to be used in conditional branching. At the same time the signal *useCarry*, generated by the Instruction Unit, indicates the ALU whether to use or not the previous carry as carry in for the present operation.

As can be seen from Fig. 36, the ALU is split into 3 fixed computational sections (Arithmetic Unit, Logic Unit and Shift Unit), some external computational units (only one is shown in the figure) and then a data selection path and registering section described below.

The signal *useCarry* is fed into the modules than may generate a carry out, thus would use a carry in. It indicates whether to use the previous registered carry as carry in or not.

Along with the partial results, each operational unit sends the carry out bit to the mux to be registered out of the ALU. It is indicated in the diagram by C_ in front of the partial result name. This bit is concatenated with the result, becoming the words one bit wider than normal. In the case of the Logic Unit and the External Units that do not generate carry bits, the value of this bit is zero. The external units connected to the EVE ALU may or may not generate carry out. The carry in those cases will be treated as in the normal blocks.

The mux passes through the result and the corresponding carry and then they are registered out separately. Finally, the signals coming out of the ALU block are *aluRes* and *carry*.

The Adder Unit performs the arithmetic functions of the processor i.e. the addition and subtraction. Also included in this section are the comparison functions. The comparison instructions are implemented as subtractions. The result is then compared to see if it is positive, negative or equal to zero, depending on the comparison carried out.

It has two main components, an adder and a control logic block. The Adder performs a simple addition of two operands and a carry bit. This carry could be the carry out from the previous instruction or a bit set when subtracting. It means that the carry out can only be used when adding, but not when subtracting or performing a comparison operation.

If a subtraction, whether it is a subtract instruction or a comparison instruction, is to be performed, the signal *selS2* is set high which selects an inverted version of the second operand and is added to the first operand, along with a '1'. This is, effectively, the addition of the 2's compliment of the number that is to be subtracted.

The flow diagram, Fig. 37, shows the control logic of the adder unit. Depending on the operation to be performed the control logic block selects the correct S2 operand, whether inverted or not. It then does simple comparisons as needed. Fig. 38 shows the flow diagram of the Adder unit.

The processor implements 3 logic functions, AND, OR and Exclusive OR. Fig. 39 shows the implementation of this.

Implemented in the Shifter Unit are the following three instructions:

- Shift Left Logical (SLL). This shifts the data on *aluS1* to the **left** by the number of times indicated by the data on *aluS2*. A logical shift to the left places zeros in the least significant bit position of the data being shifted.

- Shift Right Logical (SRL). This shifts the data on *aluS1* to the **right** by the number of times indicated by the data on *aluS2*. In a logical shift to right zeros are placed in the most significant bit position of the data being shifted.

- Shift Right Arithmetical (SRA). This shifts the data on aluS1 to the **right** by the number of times indicated by the data on aluS2. This is an arithmetic shift meaning that as the data is shifted to the right, the value of the bit that was in the most significant bit position is the value of the bit that is shifted in, primarily to maintain the correct sign of the integer being shifted.

The processor is adapted for connection to Custom External Units that perform specific operations. They will be selected by *aluOp*. These operations can last more than one clock cycle, so they need a mechanism to stall the pipeline until they are completed. It is implemented by the signal *resValid*, which is asserted by the active unit in order to stall the pipeline until the operation is concluded, when the signal is deasserted and the pipeline released.

If none of the External Units requires the use of *resValid*, it must tied to zero.

Data being stored in memory and the fact that it passes a different path through the Execution Unit was mentioned above. The size of the data being stored can be of 3 sizes, word, halfword and byte. Fig. 40 shows the flow of data through the replication logic, assuming a 32 bit processor. It can be extended to n bits, but for simplicity and ease of understanding 32-bits is used.

Taking the case of a byte being written to memory. The byte to be written will be in the bottom byte position of the register it is being held in, however, the Instruction Unit and Execution Unit cannot determine what byte position within the word that the byte has to be placed. This is determined by the address that is generated in the ALU. This appears too late in the clock cycle when the Store Byte instruction is in the Execution stage. Therefore, it is more reasonable to replicate this byte up through the rest of the word so that it appears in all possible byte locations. When the data to be written to memory is in the Execution stage of the pipeline, it has quite a lot of time before it is registered through to the Memory stage, where it will have much less time to be manipulated, before it is written to memory. Therefore,

this replication can happen during the Execution stage by the Execution Unit.

The same situation applies to halfwords being written to memory by a Store Halfword instruction except that it is only replicated once so that it appears in the upper half. Finally, with the Store Word instruction, there is no replication required and so the data just passes straight through.

There are two main points of data selection, the first is operand sources for the ALU and data memory data sources and the second is ALU result selection. Within the Source Operand Muxes block, the ALU operands and memory data are selected. Selecting the correct data sources with the Execution Unit are done by multiplexers in the Source Operand Muxes block, shown in Fig. 42.

Firstly, the selection of operands for the ALU is separated into the two operands. As can be seen from the block diagram, common to both are the two forwarded data busses, *aluRes* and *dataBack* as data sources. For the **S1Mux** the other two operands are *S1* which is the data registered through from the selected S1 register in the Decode stage and the Program Counter value of the instruction following the instruction which is at this execution stage of its cycle. For the **S2Mux** the other two operands are *S2* which is the data registered through from the selected S2 register in the Decode stage and an immediate operand that has been generated from the contents of the instruction word. The two select busses for these muxes are generated in the Instruction Unit. They are encoded as shown in Table 20.

The **S2PassMux** performs the selection of the data that is to be stored in memory. As with the ALU operand selection muxes, two of its data sources the forwarded busses *aluRes* and *dataBack*. The third source is *S2* which is the data registered through from the selected S2 register in the Decode stage.

Table 20: Source Muxes select lines encoding values

Code	*s1MuxSel*	*s2MuxSel*	*s2PassMuxSel*
00	REGS1	REGS2	PMREGS2
01	PCDATA	IMMDATA	unused
10	EXDATA1	EXDATA2	PMEXDATA2
11	MEMDATA1	MEMDATA2	PMMEMDATA2

The second area of selection is the result of the ALU operation performed. This selects from 3 fixed units of the ALU, the external units if present and also the fed back data from the previous operation when a processor stall is happening as can be seen in Fig. 36. A small block of logic generates the control for this mux, as shown in Fig. 42 below. The diagram shows the basic configuration, it is without external units.

The Data Unit 64 elements comprise the Memory stage of the processor. They are the data memory Controller, the data memory Sign Extend Unit and the Memory stage Registers. Fig. 43 shows those blocks and the signals that interconnect themselves and with the other units of the EVE implementation

The Memory Controller receives two control signals from the previous stage, *dmCS* and *dmCtrl*. When the signal *dmCS* is asserted, the Memory Controller generates the control signals for the data memory, the data memory Sign Extend Unit and the Data Muxes according to *dmCtrl*. The encoding values for *dmCtrl* are listed in Table 21. The Memory Controller flow diagram is illustrated in Fig. 44.

Table 21 - Encoding values for *dmCtrl* signal.

Code	Label	Comments
0 00 0	STBYTE	Store byte
0 00 1	Unused	
0 01 0	STHALF	Store halfword
0 01 1	STWORD	Store word
0 1x x	Unused	
1 00 0	LDBYTEUN	Load byte unsigned
1 00 1	LDBYTESIG	Load byte signed
1 01 0	LDHALFUN	Load halfword unsigned
1 01 1	LDHALFSIG	Load halfword signed
1 10 0	LDWORD	Load word
1 10 1	Unused	
1 11 0	Unused	
1 11 1	Unused	

The bit 3 of *dmCtrl* indicates the operation (load or store), bits 2 and 1 indicate the size of the memory access (byte, halfword or word) and bit 0 indicates if the operation is signed or unsigned.

The control signals generated for data memory are *dmRW*, which depends on the access being caused by a load or a store instruction and *dmSize* that indicates if the data contains a byte, a halfword or a word.

The control signal for the mux that chooses the data registered to the next stage of the pipeline depends on the type of instruction. When loading, the mux passes the data coming out from data memory and when storing, it chooses ALU result.

Finally, this sub-block generates a signal, *dmSESel*, to select which kind of sign extension has to be performed on the data coming from memory. It is asserted when loading a byte or a halfword, and also depends on the type of load being performed (signed or unsigned). Otherwise, data coming from memory goes through this sub-block without being changed. The encoding values are shown in Table 22.

Table 22 - Encoding values for dmSESel signal

code	Label	Comments
0xx	PASSTHROUGH	Pass word through
100	ZEXTBYTE	Zero Extend byte
101	SEXTBYTE	Sign Extend byte
110	ZEXTHALF	Zero Extend halfword
111	SEXTHALF	Sign Extend halfword

The data memory Sign Extend Unit not only performs the sign extension indicated by *dmSESel*, but also places the correct byte or halfword coming from data memory into the register. That operation depends on the signal *endian*, which indicates if the system is accessing data memory in little endian mode (*endian* = 1) or big endian mode (*endian* = 0). The flow diagram for the block is shown in Fig. 45, where a 32 bit data path is assumed.

Once the size (byte or halfword) is determined, the signal endian is checked in order to pick the correct part of the data and place it in the register. Then, the corresponding byte or halfword is either sign extended or zero extended. If the size indicates word, the content of the memory position addressed is just placed in the register, as it passes through this module without suffering any transformations.

The third block of the Data Unit is the Memory stage Registers, where all the signals are effectively registered into the following stage. It has three muxes in front of the corresponding registers. Two of them pass the write enable signal (*writeRegEn*) and the destination register address (*destRegAddr*) to the Write Back stage. They are controlled by the signal *memStgStall*, which indicates whether the stage has to stall by choosing the previous value registered.

The third mux chooses data from data memory (*adjDMDataIn*), ALU result (*aluRes*) or previous value (*dataBack*). Accordingly, it is not only controlled by memStgStall but also by a signal coming from Memory Controller called *loadSel*. The way the mux passes data to be registered to the Write Back stage is summarised in Table 23.

Table 23 - Data mux control signals

MemStgStall	*loadSel*	Signal Passed	Comments
1	x	*DataBack*	Write Back stage stall
0	0	*AluRes*	Default option
0	1	*AdjDMDataIn*	When loading

The Register unit 63 is built up of 3 sections. Fig. 46 shows an overview of the Register Unit 62 and its main components. The Register Files themselves are separated into two banks, the General Purpose Registers and the Special Registers. All registers are synchronous to the system clock *sysClk*.

Within the General Purpose Registers (GPRs) there are 32 addressable registers. There are in fact only 31 registers, each of these being 32-bit wide, with register 0 not being made up of actual registers but is a constant 32-bit zero. A block diagram of the GPRs is shown in Fig. 47, which breaks down into 3 sections.

By means of the parameterisation of the EVE architecture, the GPR block can be outside of the processor. In that case the inputs to the block will be driven to that external block and its outputs will be connected to the corresponding inputs in the Register Muxes block to be chosen as source operands if selected.

The signal *dataBack* returns to the registers in what is the Write Back stage of the processor pipeline. It contains the new data to be written to a register by an instruction.

This demultiplexing is controlled by the *addrDestReg* bus that contains the address of the destination register and the write enable signal, *writeRegEn*. By ANDing this write enable signal with the inverse of bit 5 of the destination register address creates a select for the GPRs. Each of the remaining bits of the destination register address are ANDed with this generated enable signal and if the enable is not set this will cause a write to register 0 which does not store a value.

The Register Block is implemented just as registers. Note that these must maintain their value on every clock period.

When an instruction addresses a register so as to use its contents, it puts a 6-bit address on one of two busses, to set this data up as operand 1 or operand 2 or both. All but the top bits of these two busses drive multiplexers that select the register value, as seen in Fig. 47 above.

The Special Registers allow up to 32 addressable registers, where the first four are always present as they keep specific processor information. These four registers are the Interrupt Register at binary address 100000, the Reason Register at binary address 100001, the Exception Address Register (EAR) at binary address 100010 and the Link Address Register (LAR) at binary address 100011. The bit field definitions of these 4 registers have been described above.

The Exception and Link Address Registers are just written to as normal through the pipeline and does not need any special logic around it. The Reason Register and the Interrupt Register, however, require further logic to handle resets and external interrupts as they happen.

While those four registers are always inside the processor, the rest of the special registers can be either inside or outside it. The placement of the registers is determined by the parameterisation.

In the case of the registers being outside the processor, the outputs of the register

bank go to the Special Registers module shown in Fig. 46 above. It is to supply the right addressed Special Register to the Register Muxes.

This register holds two bits of information, the enable bit and the pending bit. So, it must react to the resets and the interrupt control signals. A rising edge must be detected on *extint*, the signal from the external interrupting source. It will set the pending bit. This must be maintained until either an internal acknowledge or a reset has been received. At this point an acknowledge must also be given back to the external interrupting source. The processor can read from this bit, but not write to it.

On the other hand, the enable bit can be read from and written to by the processor. When this bit is set, and incoming interrupt will be serviced, otherwise it will be ignored.

The Reason Register has to show the present state of the processor and cannot wait for the latency of data passing through the pipeline. It has to react to a hardware reset *sysReset*, an illegal instruction *sReset* and either a Trap instruction or an Interrupt being serviced.

The exception address register will keep the PC value at the clock cycle a TRAP or an interrupt changes the program order to be serviced. The instruction corresponding to this PC value is annulled, so its execution has to be restarted once the Exception Routine is finished.

EAR is written by the though the pipeline when a TRAP or an interrupt are detected and can be serviced. The address stored is read by the instruction RFE, which causes next PC to be loaded with it in order to fetch again the instruction previously annulled.

The link address register is used when the execution of JAL or JALR will cause a change in the program order. They change the value of PC to the target address and the annulation of the instruction just been fetched. The address of that instruction is stored in LAR.

LAR is written by the though the pipeline by JAL or JALR. Instruction RET will read the stored address, which will be loaded into next PC to restart the execution of the

instruction previously annulled.

There are only two operands required by the Execution Unit and either a General Purpose register or a Special register can be selected at any one time for each of these operands. The data returned to a specific register may be needed in the next clock period before it can be written back into the register, therefore, a forwarded path of *dataBack* is required.

The case of a stall occurring also needs to be covered where the data in the previous clock period needs to be sent through again. The block diagram in Fig. 48 shows the two muxes that perform the selection of the data as the sources for the Execution Unit. The control for these muxes is handled in the Instruction Unit, where the instruction is decoded and thus the decision of what data to use is implemented. The data selected is then registered out of this pipe stage and into the Execution stage.

The inputs regS1 and regS2 are driven by the outputs of the General Purpose Registers, either being placed inside or outside. This fact is reflected in the block instantiation, where the actual signals are connected to the formal signals of the block.

The situation of the signals sRegS1 and sRegS2 is different. They always come from the Special Registers block independently of part of the registers being inside or outside.

In the specification the terms "comprise, comprises, comprised and comprising" or any variation thereof and the terms "include, includes, included and including" or any variation thereof are considered to be totally interchangeable and they should all be afforded the widest possible interpretation and vice versa.

The invention is not limited to the embodiment hereinbefore described, but may be varied in both construction and detail within the scope of the claims.

CRUICKSHANK & CO.

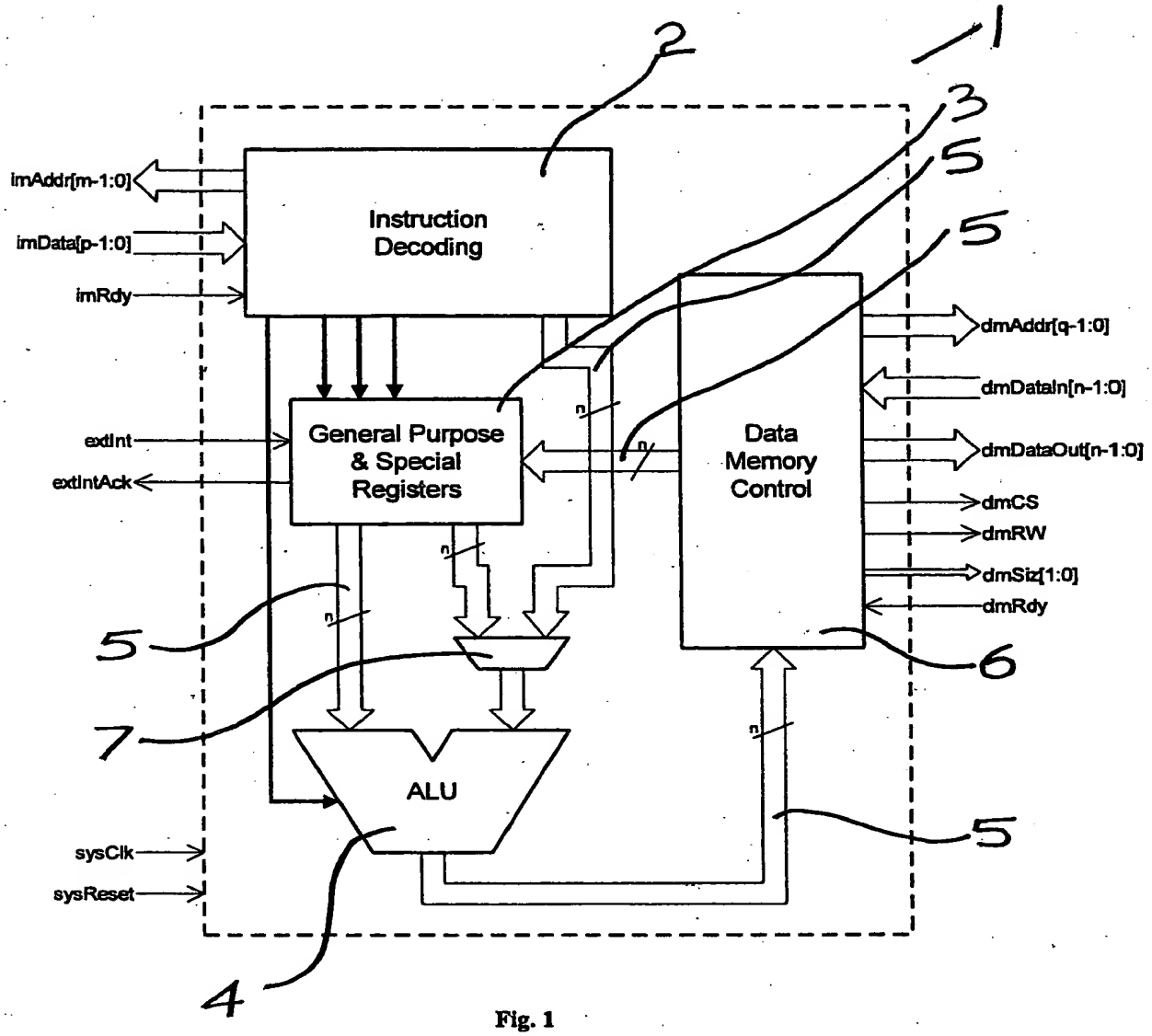

1
2
3
5
5
5
5
6
7
4
imAddr[m-1:0]
imData[p-1:0]
imRdy
Instruction Decoding
extInt
extIntAck
General Purpose & Special Registers
Data Memory Control
dmAddr[q-1:0]
dmDataIn[n-1:0]
dmDataOut[n-1:0]
dmCS
dmRW
dmSiz[1:0]
dmRdy
ALU
sysClk
sysReset
n

Fig. 1

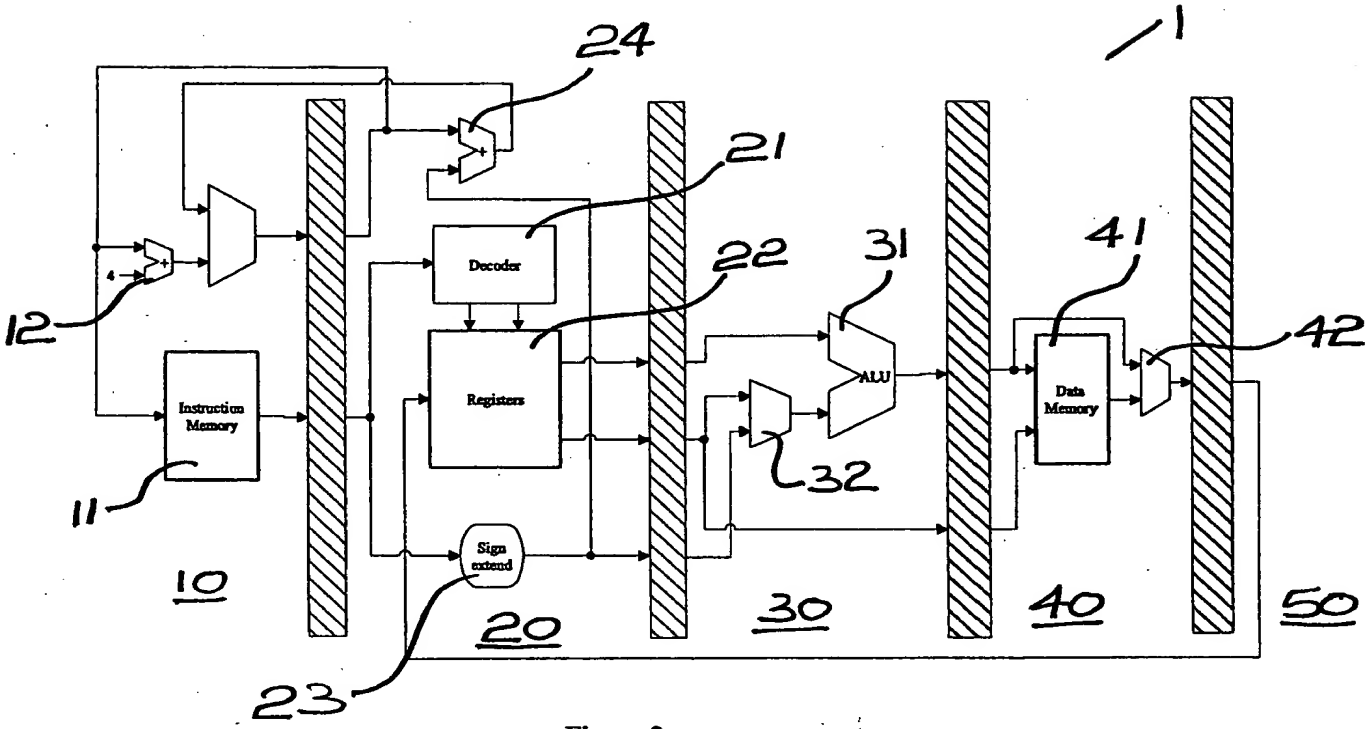

1
24
21
22
31
41
42
12
Decoder
Registers
ALU
Data Memory
Instruction Memory
11
32
Sign extend
23
10
20
30
40
50

Figure 2

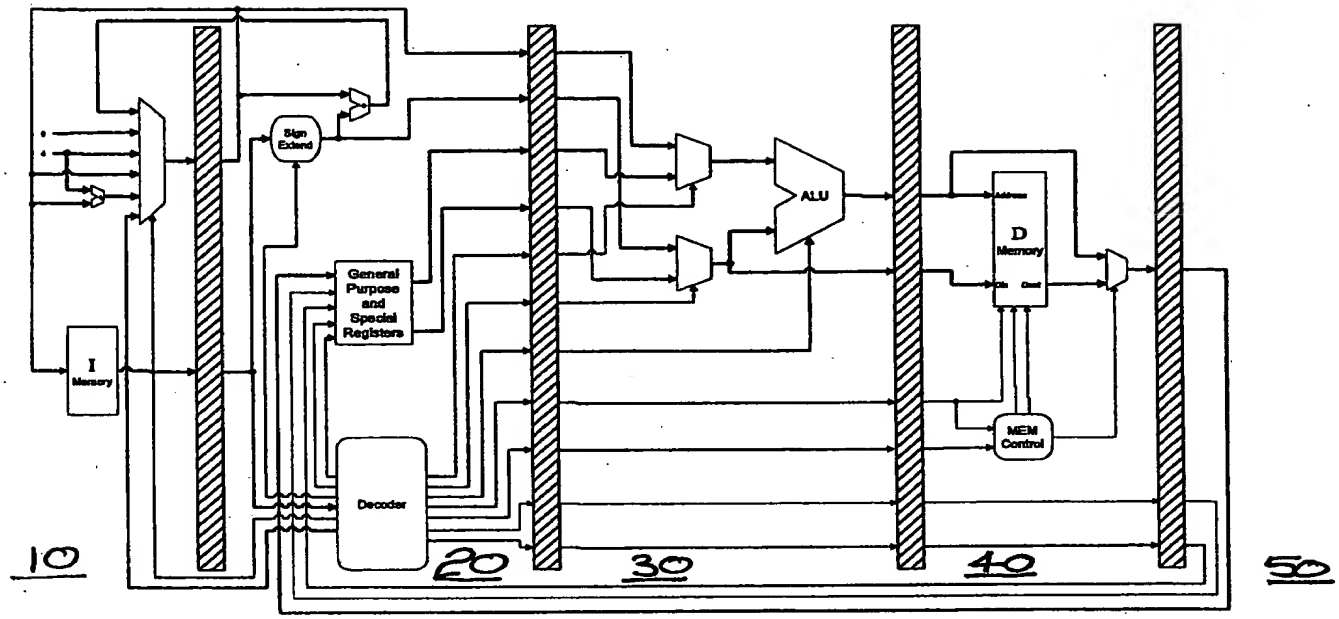

Sign Extend
General Purpose and Special Registers
I Memory
Decoder
ALU
D Memory
MEM Control
10
20
30
40
50

Fig. 3

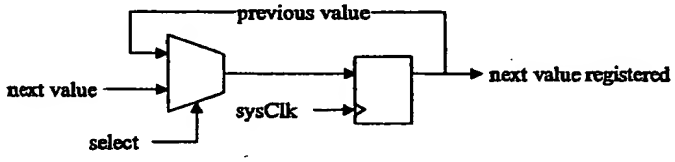

previous value
next value
select
sysClk
next value registered

Fig. 4

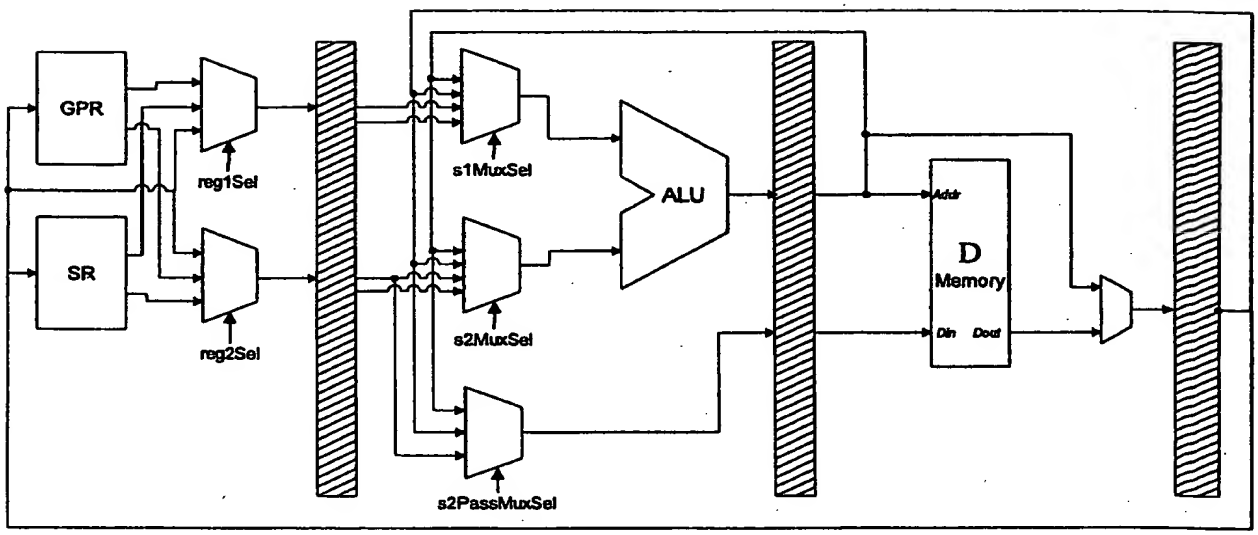

GPR
SR
reg1Sel
reg2Sel
s1MuxSel
s2MuxSel
s2PassMuxSel
ALU
Addr
D
Memory
Din
Dout

Fig. 5

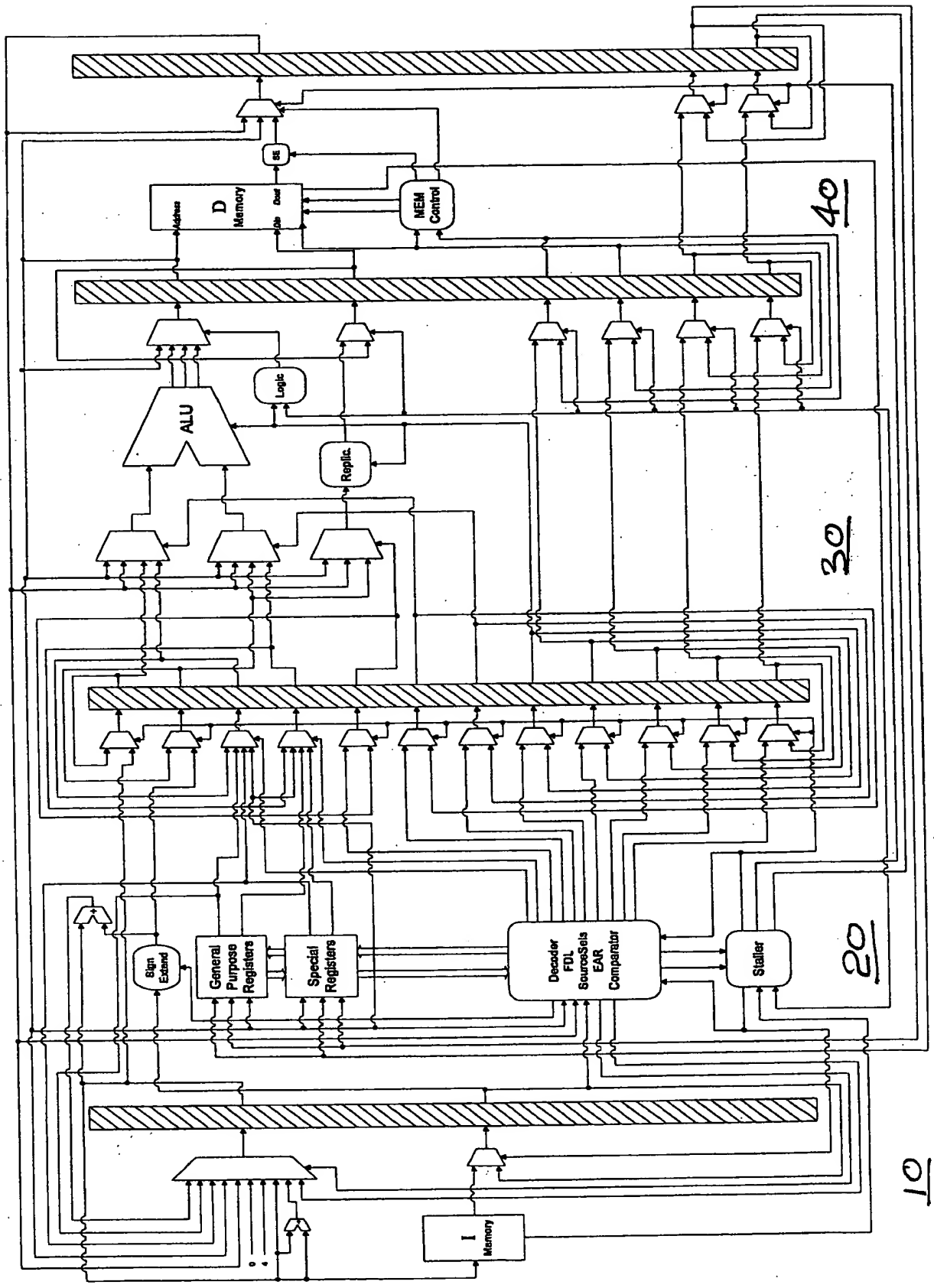

81
40
30
20
10
D Memory
MEM Control
Logic
ALU
Replic.
Sign Extend
General Purpose Registers
Special Registers
Decoder FDL SourceSels EAR Comparator
Staller
I Memory

Fig. 6

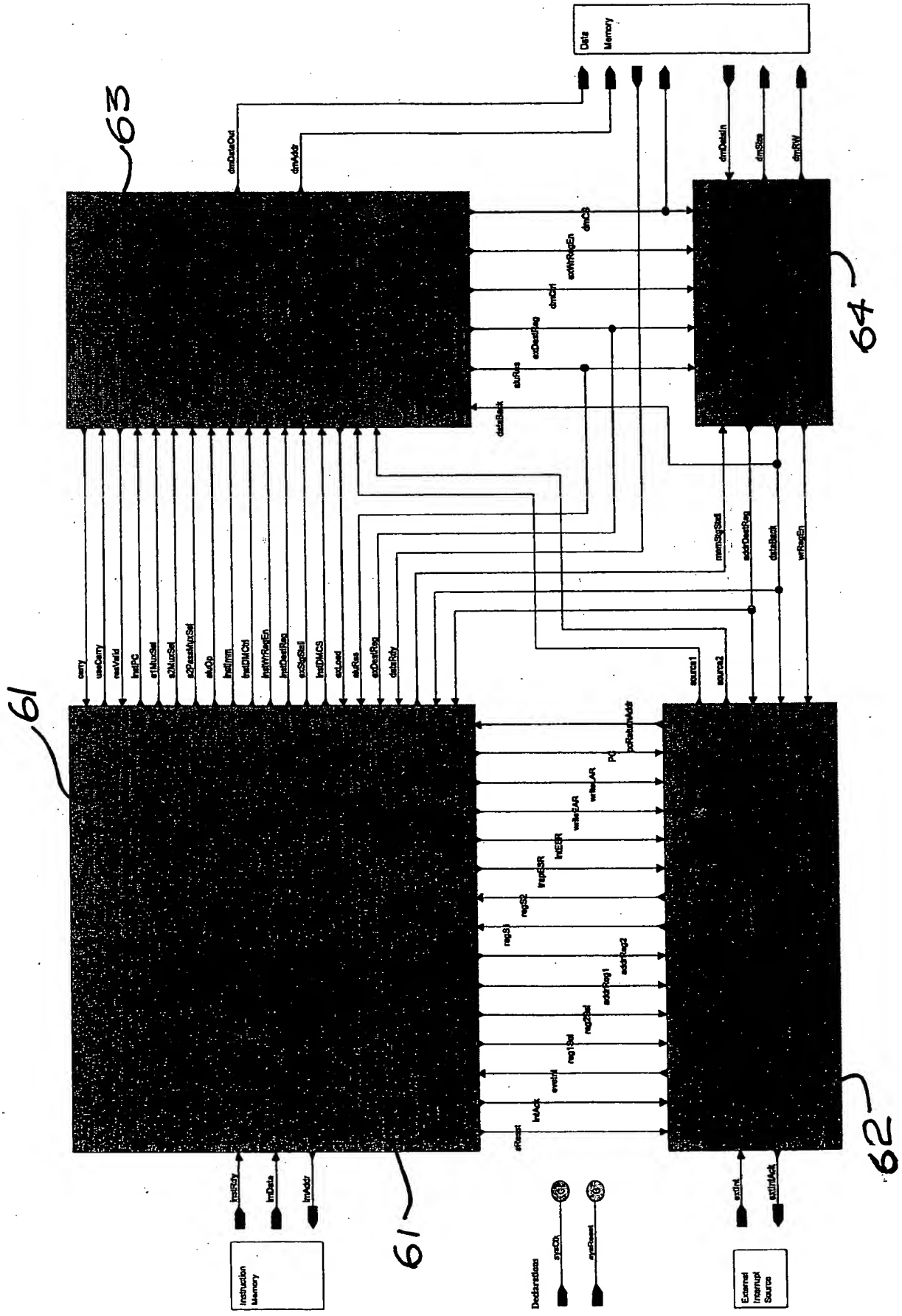

Data Memory
63
64
61
62
dmDataOut
dmAddr
dmDataIn
dmSize
dmRW
dmCtl
exWrReqEn
exDestReg
aluRes
dataBack
carry
useCarry
resValid
instPC
s1MuxSel
s2MuxSel
s2PassMuxSel
aluOp
instImm
instDMCtl
instWrRegEn
instDestReg
exSigSel
instDMCS
exLoad
aluRes
exDestReg
dataRdy
source1
source2
mmSigSel
exDestReg
dataBack
wrRegEn
pcRetAddr
PC
writeLAR
writeSAR
intEER
trapSNR
reg32
reg31
addrReg2
addrReg1
reg2Sel
reg1Sel
eventi
intAck
eReset
instRdy
inData
inAddr
Instruction Memory
Declarations
sysClk
sysReset
extInt
extIntAck
External Interrupt Source

Fig. 7

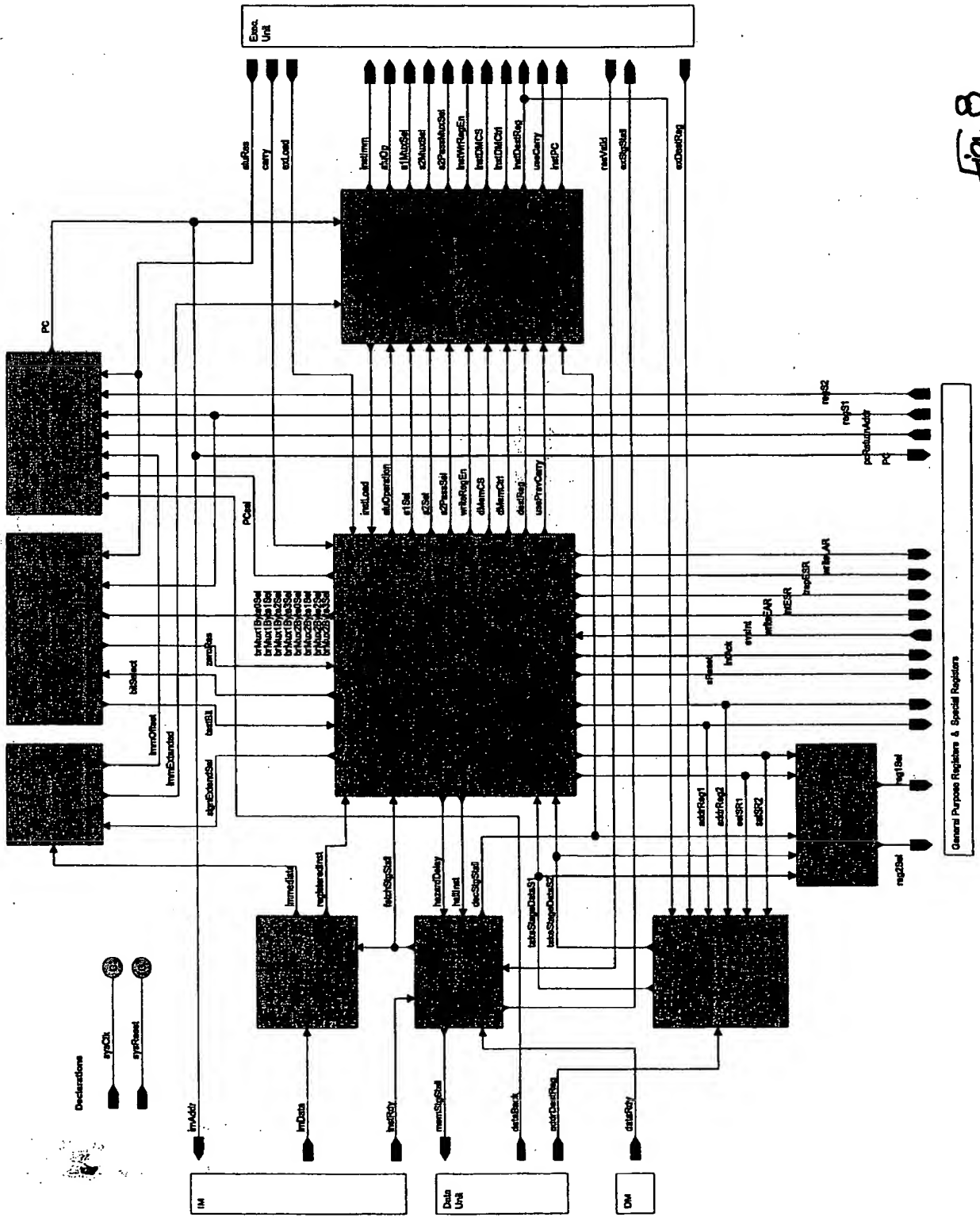

Exec. Unit
aluRes
carry
extLoad
instImm
aluOp
s1MuxSel
s2MuxSel
s2PassMuxSel
instWrRegEn
instDMCS
instDMCtrl
instDestReg
usePrevCarry
instPC
newValid
exSysStall
exDestReg
PC
PCsel
instLoad
aluOperation
s1Sel
s2Sel
s2PassSel
writeRegEn
dMemCS
dMemCtrl
destReg
usePrevCarry
brAluri8byte0Sel
brAluri8byte1Sel
brAluri8byte2Sel
brAluri8byte3Sel
brAluri20to0Sel
brAluri20to8Sel
brAluri20to16Sel
brAluri20to24Sel
brAlu28to2Sel
brAlu28to30Sel
zeroRes
bitSelect
bitSel
immOffset
immExtended
signExtendSel
immediate
registeredInst
fetchStgStall
haltInst
decStgStall
haltStageData1
haltStageData2
addrReg1
addrReg2
selSR1
selSR2
reg1Sel
reg2Sel
reg0
reg1
reg2
pcReturnAddr
PC
intMask
intClr
intInt
writeEAR
intESR
writeESR
writeLAR
sReset
General Purpose Registers & Special Registers
Declarations
sysClk
sysReset
imAddr
imData
instRdy
memStgStall
dataBank
dataDestReg
dataRdy
IM
Data Unit
DM

Fig. 8

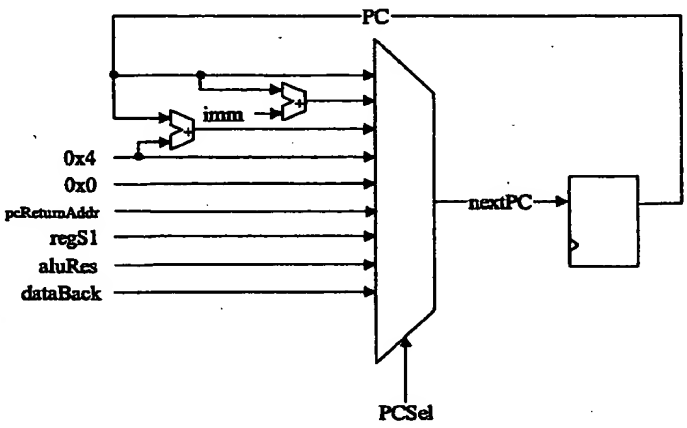

PC
imm
0x4
0x0
pcReturnAddr
regS1
aluRes
dataBack
nextPC
PCSel

Fig. 9

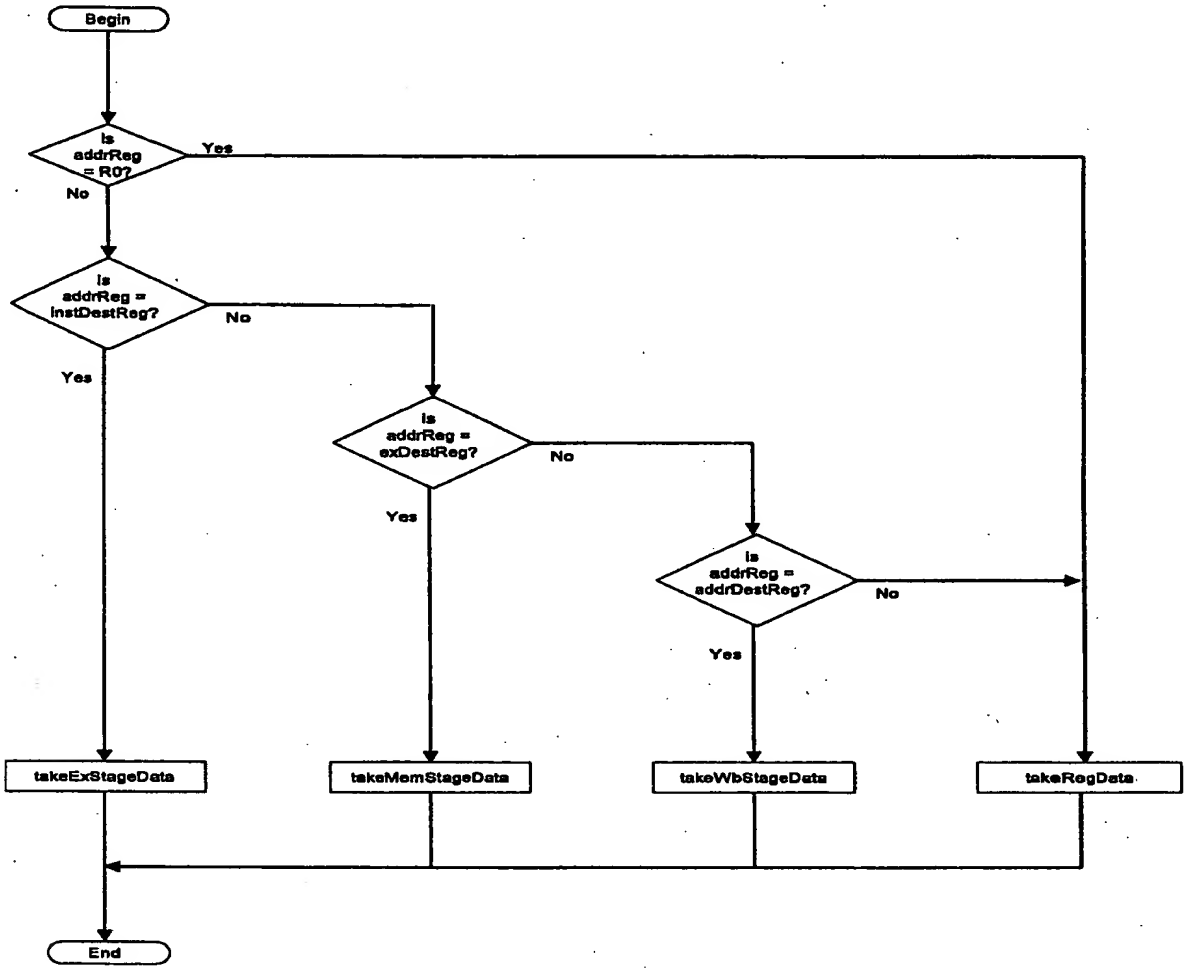

Begin
Is addrReg = R0?
Yes
No
Is addrReg = instDestReg?
No
Yes
Is addrReg = exDestReg?
No
Yes
Is addrReg = addrDestReg?
No
Yes
takeExStageData
takeMemStageData
takeWbStageData
takeRegData
End

Fig. 10

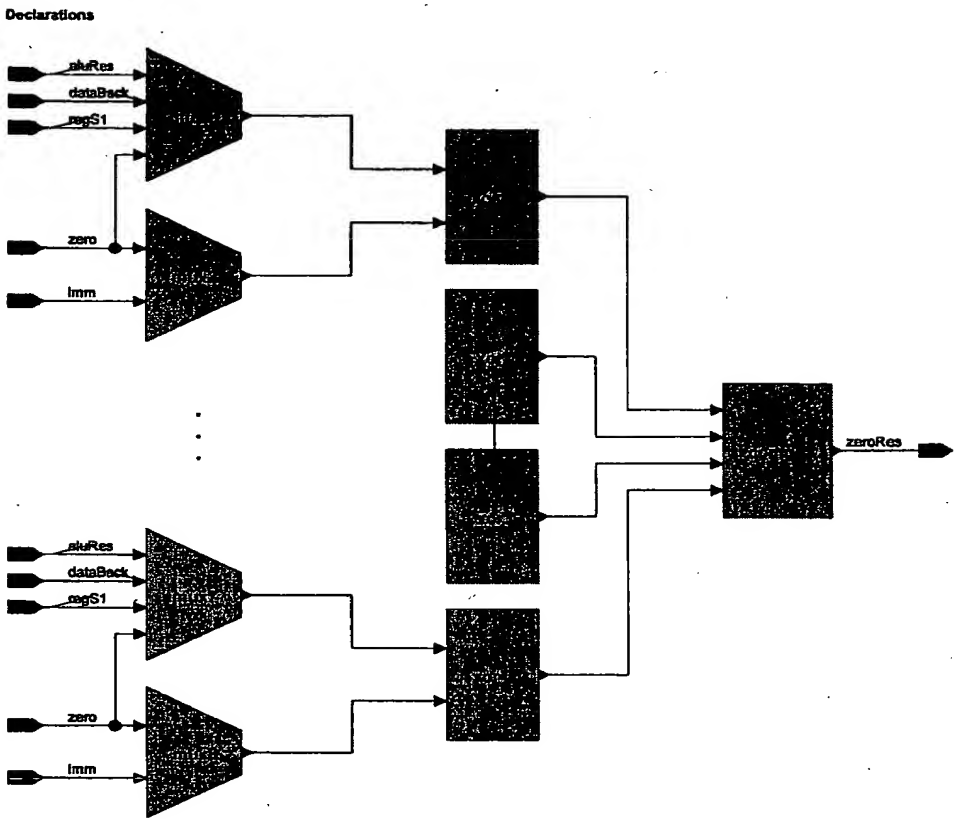

Declarations
aluRes
dataBack
regS1
zero
Imm
aluRes
dataBack
regS1
zero
Imm
zeroRes

Fig. 11

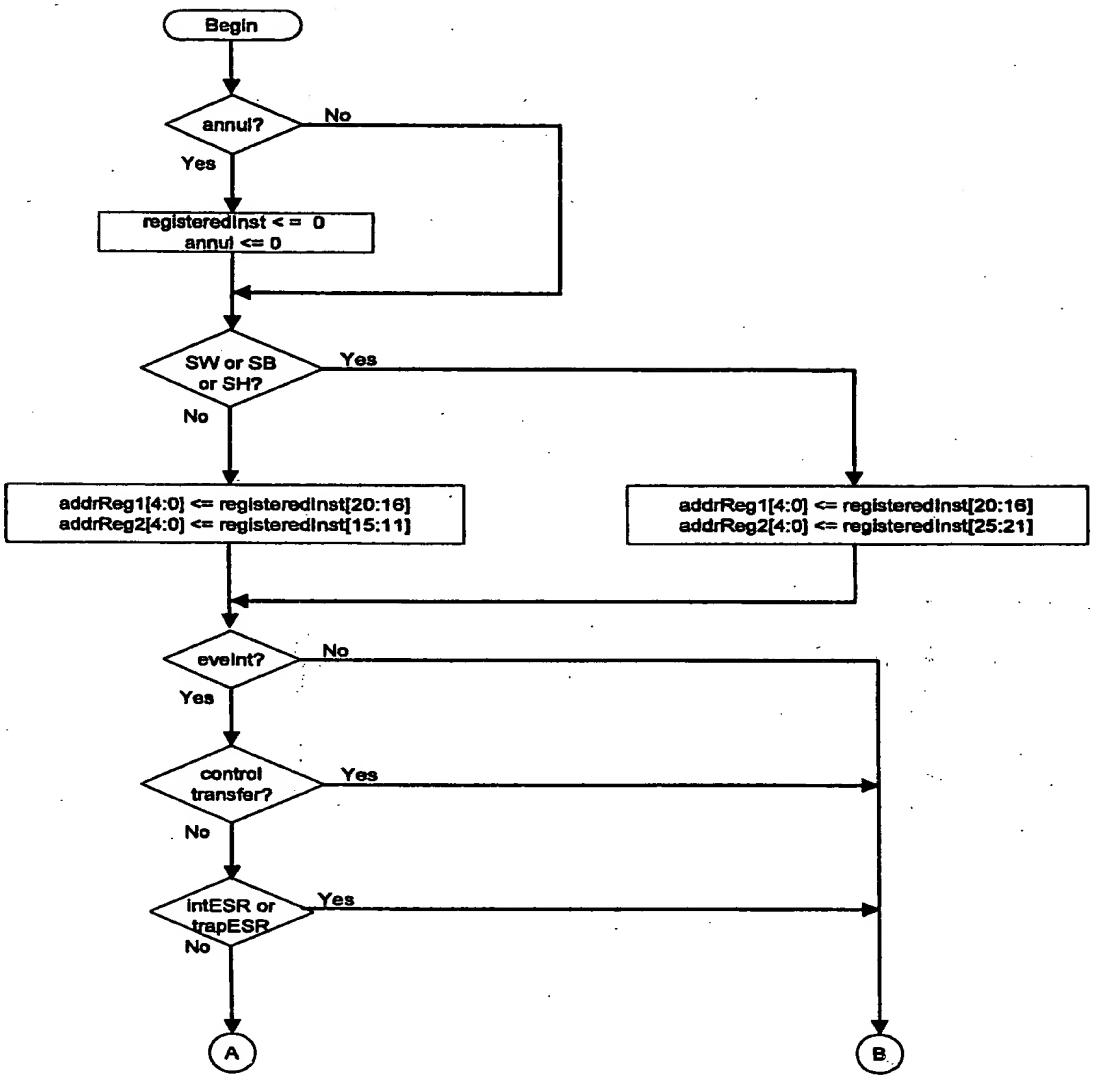

Begin
annul?
No
Yes
registeredInst <= 0
annul <= 0
SW or SB or SH?
Yes
No
addrReg1[4:0] <= registeredInst[20:16]
addrReg2[4:0] <= registeredInst[15:11]
addrReg1[4:0] <= registeredInst[20:16]
addrReg2[4:0] <= registeredInst[25:21]
eveInt?
No
Yes
control transfer?
Yes
No
intESR or trapESR
Yes
No
A
B

Fig. 12

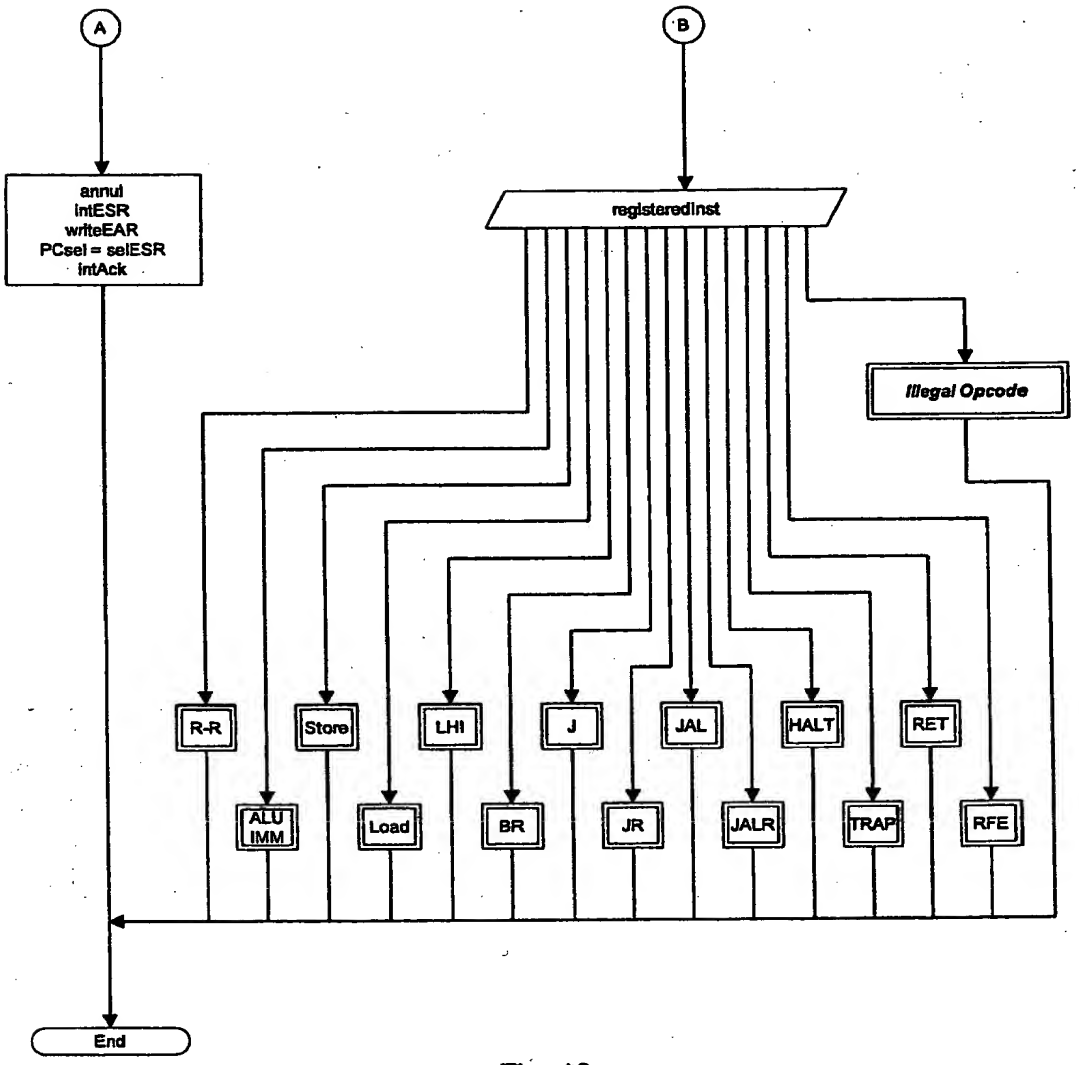

A
B
annul
intESR
writeEAR
PCsel = selESR
intAck
registeredInst
Illegal Opcode
R-R
Store
LHI
J
JAL
HALT
RET
ALU IMM
Load
BR
JR
JALR
TRAP
RFE
End

Fig. 13

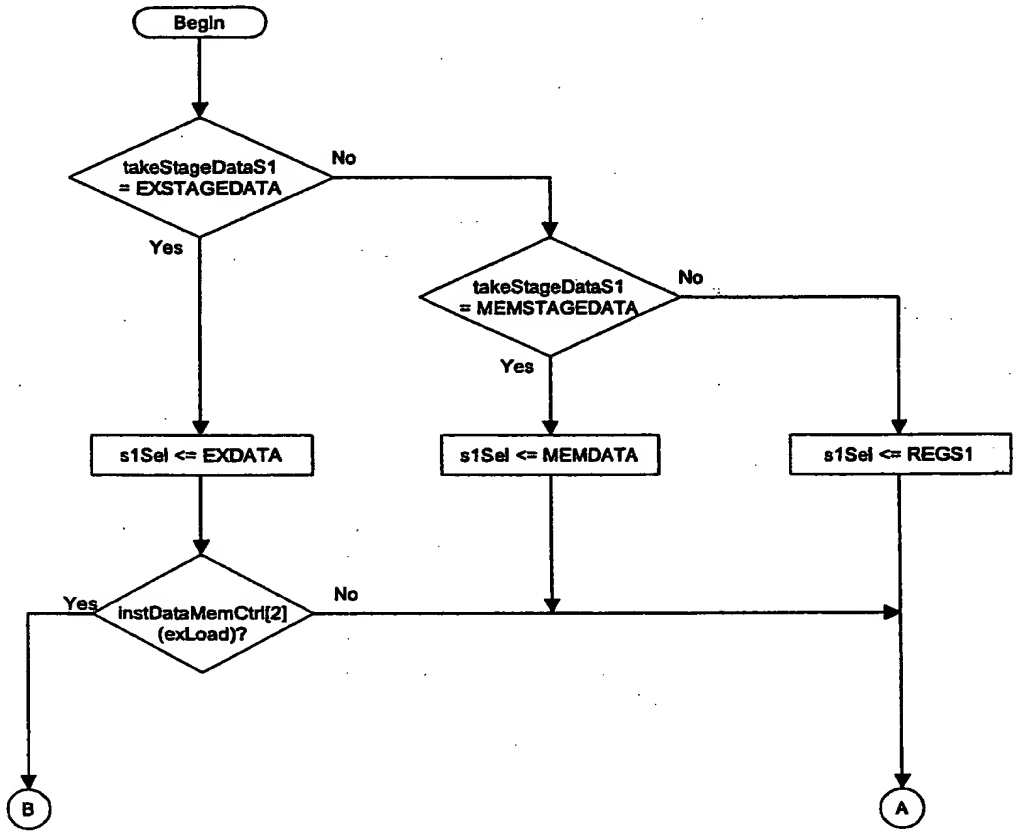

Begin
takeStageDataS1 = EXSTAGEDATA
No
Yes
takeStageDataS1 = MEMSTAGEDATA
No
Yes
s1Sel <= EXDATA
s1Sel <= MEMDATA
s1Sel <= REGS1
Yes
instDataMemCtrl[2] (exLoad)?
No
B
A

Fig. 14

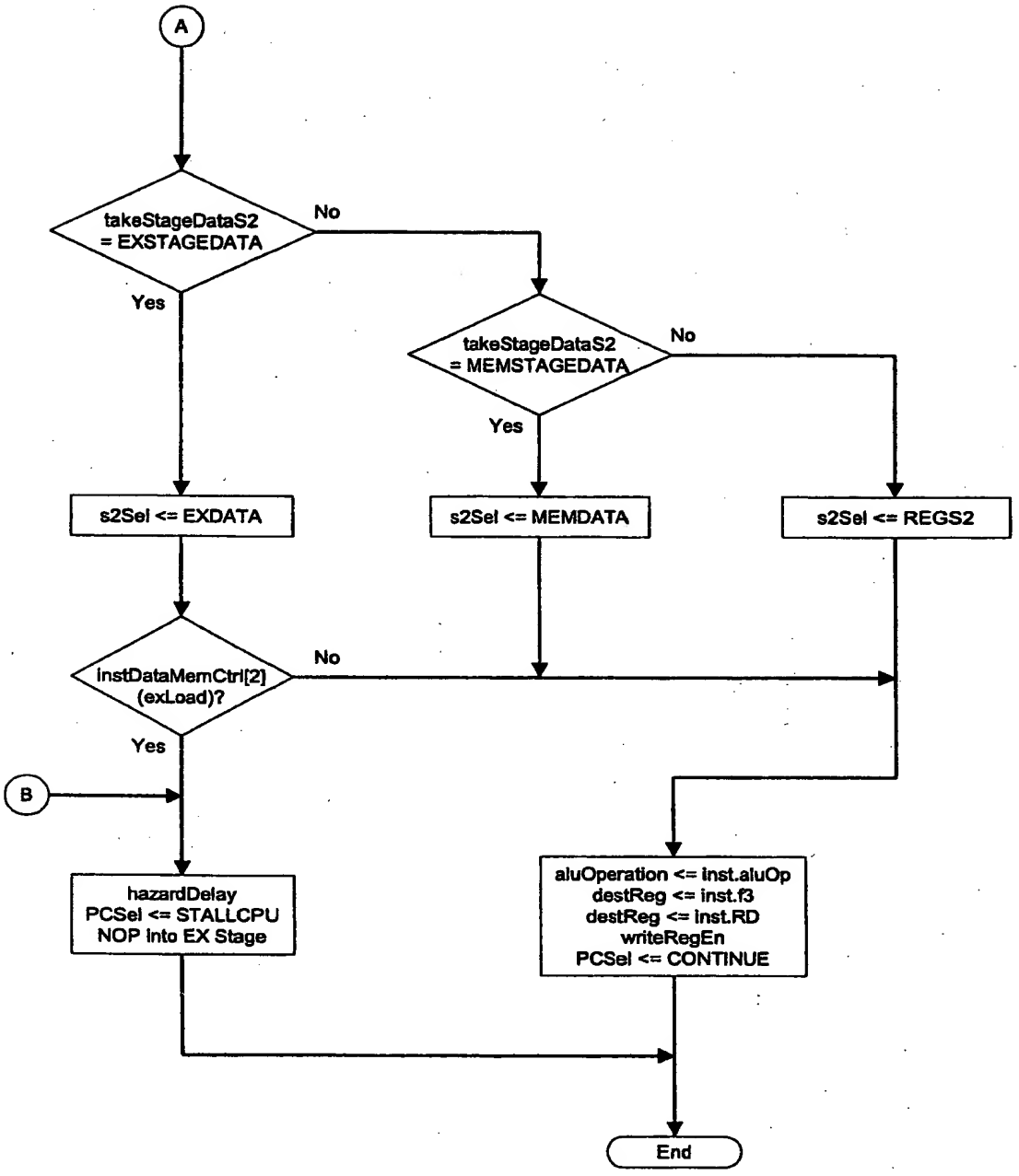

A
takeStageDataS2
= EXSTAGEDATA
No
Yes
takeStageDataS2
= MEMSTAGEDATA
No
Yes
s2Sel <= EXDATA
s2Sel <= MEMDATA
s2Sel <= REGS2
instDataMemCtrl[2]
(exLoad)?
No
Yes
B
hazardDelay
PCSel <= STALLCPU
NOP into EX Stage
aluOperation <= inst.aluOp
destReg <= inst.f3
destReg <= inst.RD
writeRegEn
PCSel <= CONTINUE
End

Fig. 15

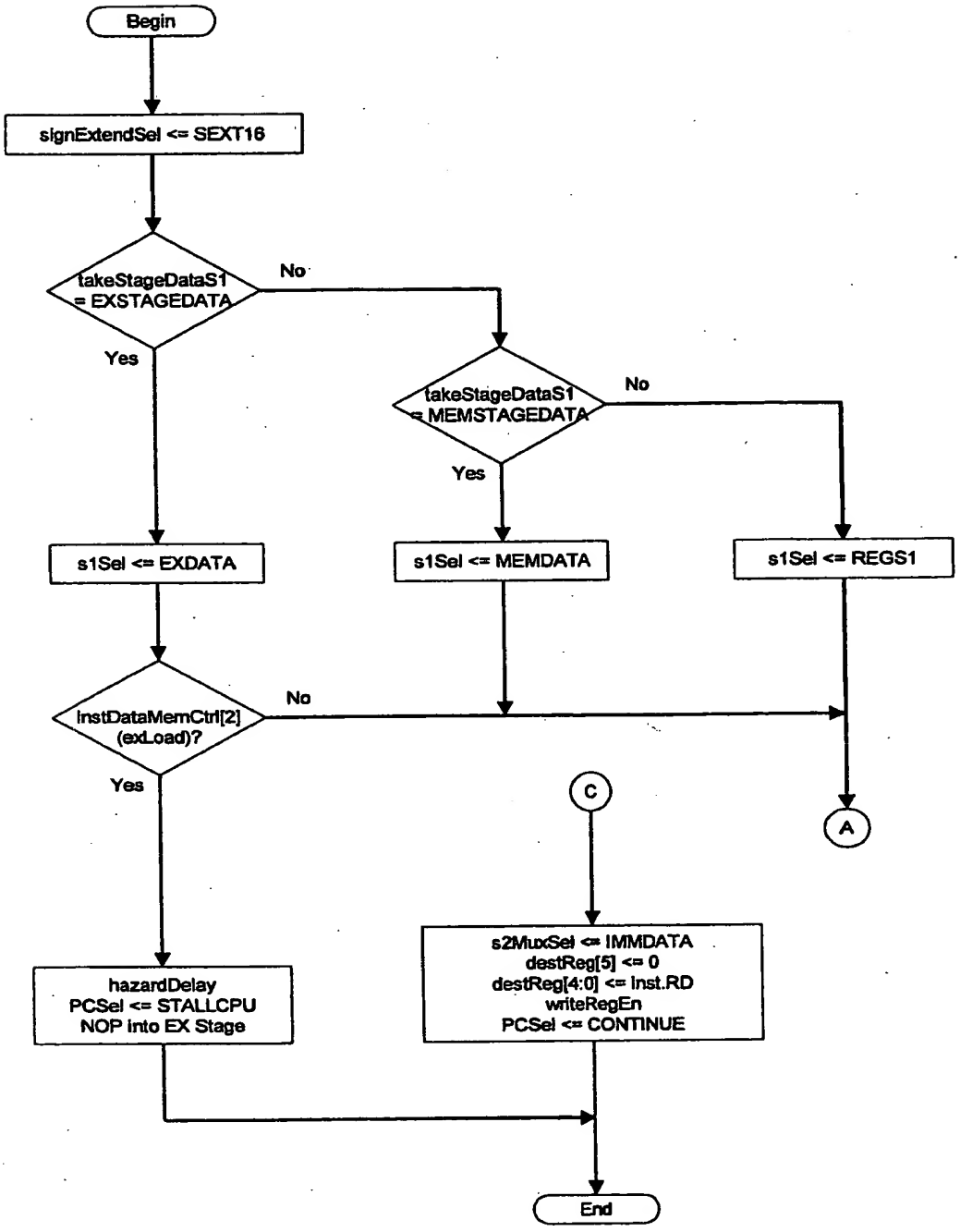

Begin
signExtendSel <= SEXT16
takeStageDataS1 = EXSTAGEDATA
No
Yes
takeStageDataS1 = MEMSTAGEDATA
No
Yes
s1Sel <= EXDATA
s1Sel <= MEMDATA
s1Sel <= REGS1
instDataMemCtrl[2] (exLoad)?
No
Yes
C
A
s2MuxSel <= IMMDATA
destReg[5] <= 0
destReg[4:0] <= Inst.RD
writeRegEn
PCSel <= CONTINUE
hazardDelay
PCSel <= STALLCPU
NOP into EX Stage
End

Fig. 16

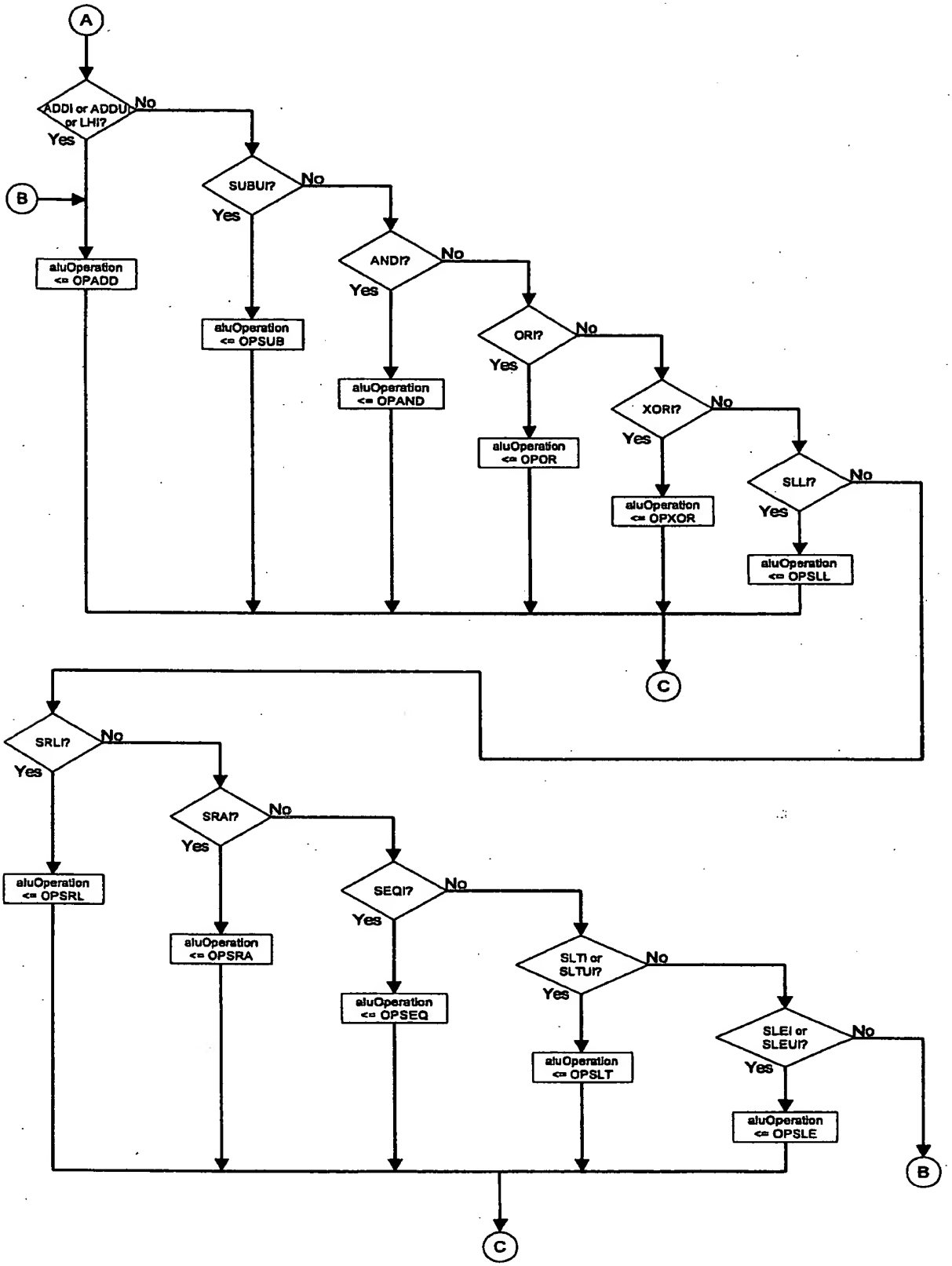

A
ADDI or ADDUI or LHI?
No
Yes
B
aluOperation <= OPADD
SUBUI?
No
Yes
aluOperation <= OPSUB
ANDI?
No
Yes
aluOperation <= OPAND
ORI?
No
Yes
aluOperation <= OPOR
XORI?
No
Yes
aluOperation <= OPXOR
SLLI?
No
Yes
aluOperation <= OPSLL
C
SRLI?
No
Yes
aluOperation <= OPSRL
SRAI?
No
Yes
aluOperation <= OPSRA
SEQI?
No
Yes
aluOperation <= OPSEQ
SLTI or SLTUI?
No
Yes
aluOperation <= OPSLT
SLEI or SLEUI?
No
Yes
aluOperation <= OPSLE
B
C

Fig. 17

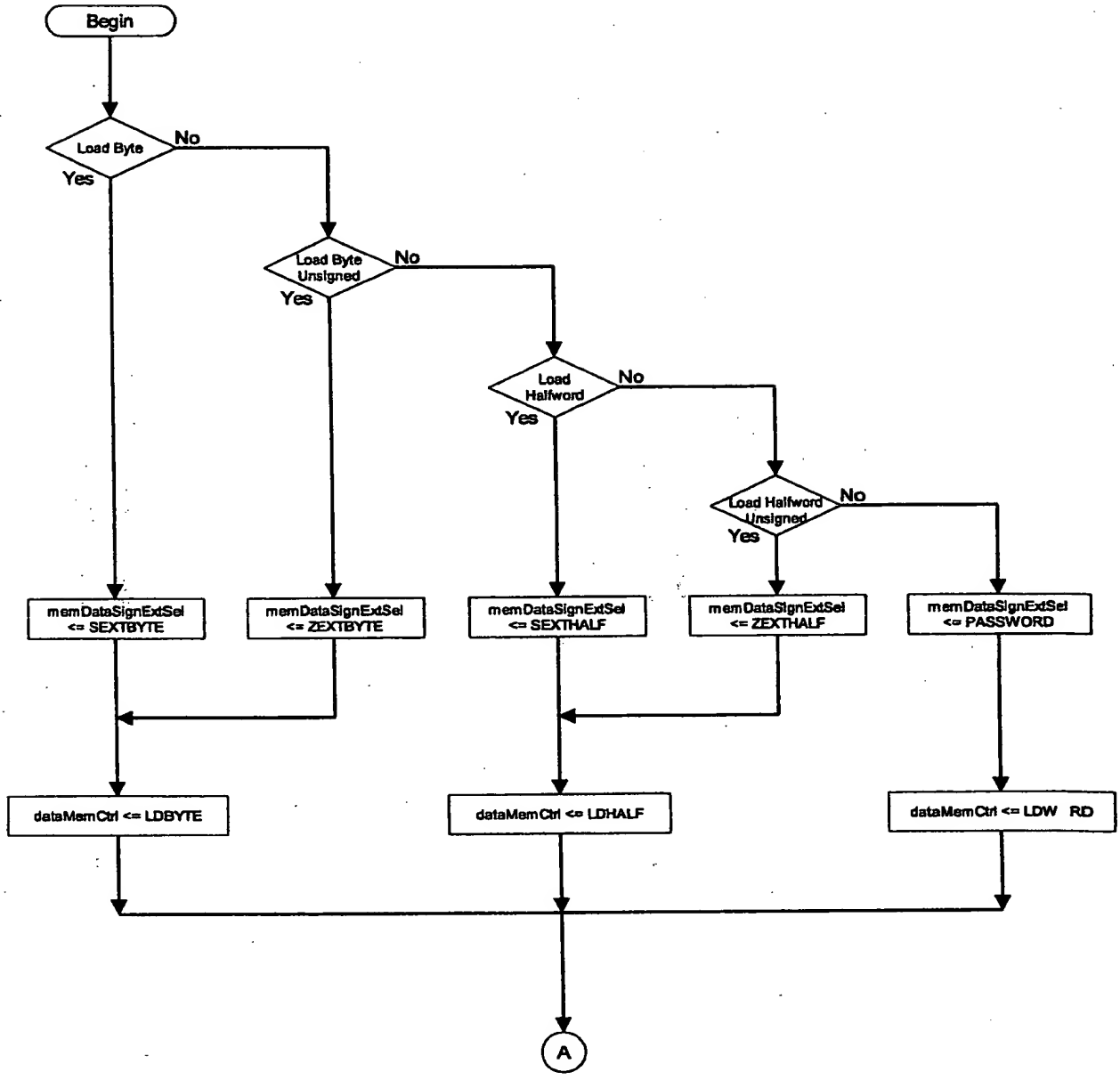

Begin
Load Byte
No
Yes
Load Byte Unsigned
No
Yes
Load Halfword
No
Yes
Load Halfword Unsigned
No
Yes
memDataSignExtSel <= SEXTBYTE
memDataSignExtSel <= ZEXTBYTE
memDataSignExtSel <= SEXTHALF
memDataSignExtSel <= ZEXTHALF
memDataSignExtSel <= PASSWORD
dataMemCtrl <= LDBYTE
dataMemCtrl <= LDHALF
dataMemCtrl <= LDW RD
A

Fig. 18

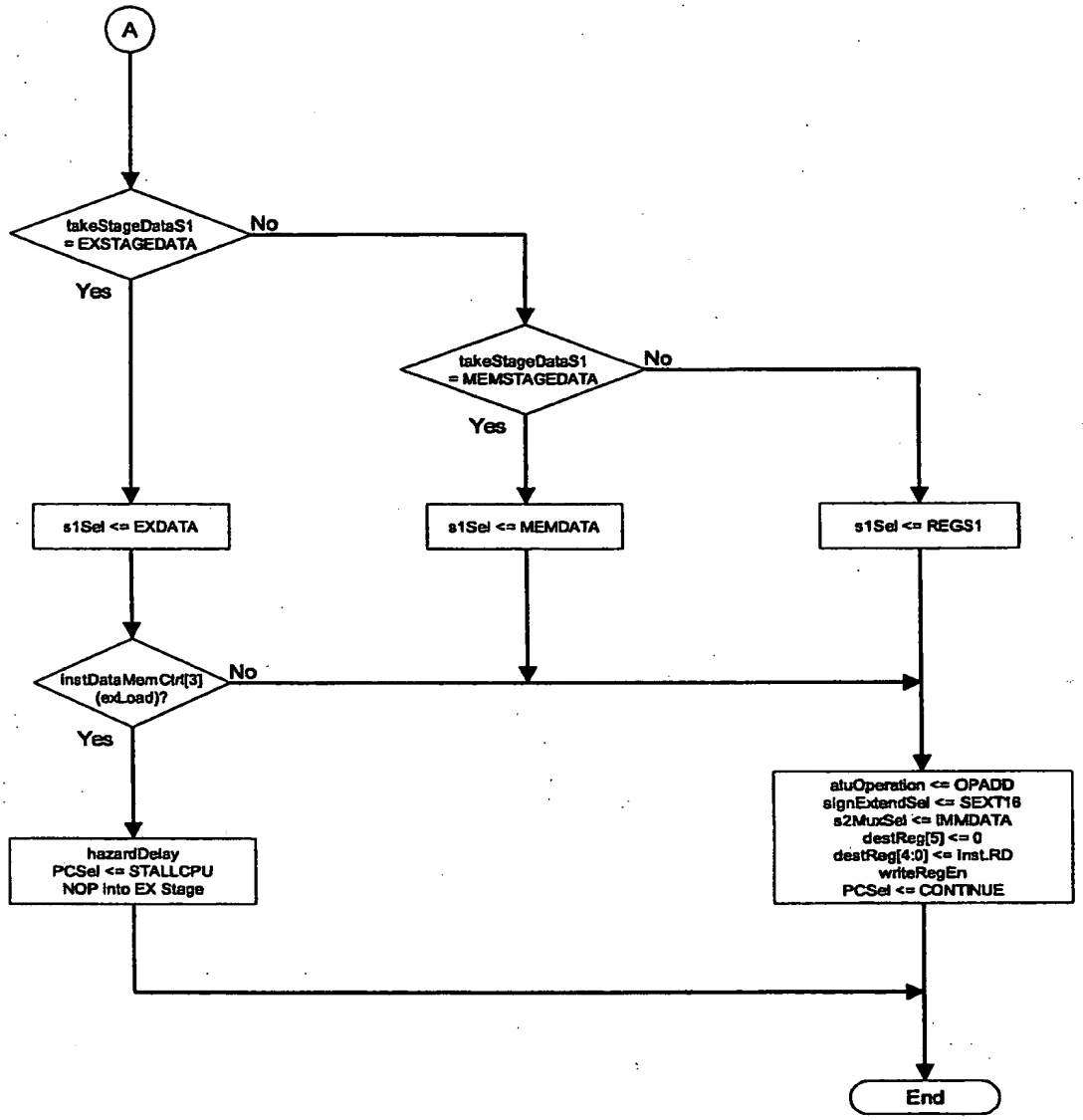

A
takeStageDataS1
= EXSTAGEDATA
No
Yes
takeStageDataS1
= MEMSTAGEDATA
No
Yes
s1Sel <= EXDATA
s1Sel <= MEMDATA
s1Sel <= REGS1
InstDataMemCtrl[3]
(exLoad)?
No
Yes
aluOperation <= OPADD
signExtendSel <= SEXT16
s2MuxSel <= IMMDATA
destReg[5] <= 0
destReg[4:0] <= Inst.RD
writeRegEn
PCSel <= CONTINUE
hazardDelay
PCSel <= STALLCPU
NOP into EX Stage
End

Fig. 19

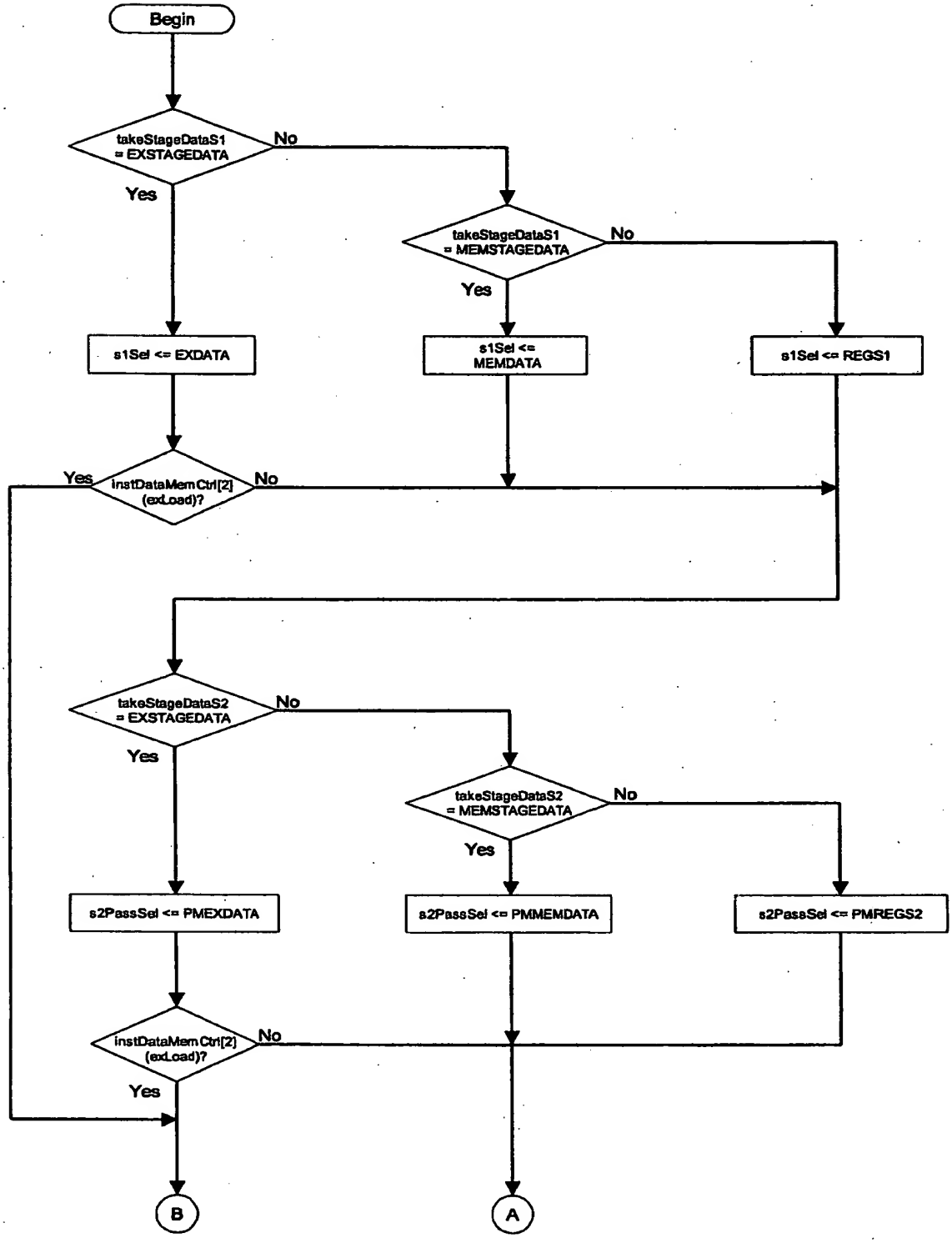

Begin
takeStageDataS1 = EXSTAGEDATA
No
Yes
takeStageDataS1 = MEMSTAGEDATA
No
Yes
s1Sel <= EXDATA
s1Sel <= MEMDATA
s1Sel <= REGS1
Yes
instDataMemCtrl[2] (exLoad)?
No
takeStageDataS2 = EXSTAGEDATA
No
Yes
takeStageDataS2 = MEMSTAGEDATA
No
Yes
s2PassSel <= PMEXDATA
s2PassSel <= PMMEMDATA
s2PassSel <= PMREGS2
instDataMemCtrl[2] (exLoad)?
No
Yes
B
A

Fig. 20

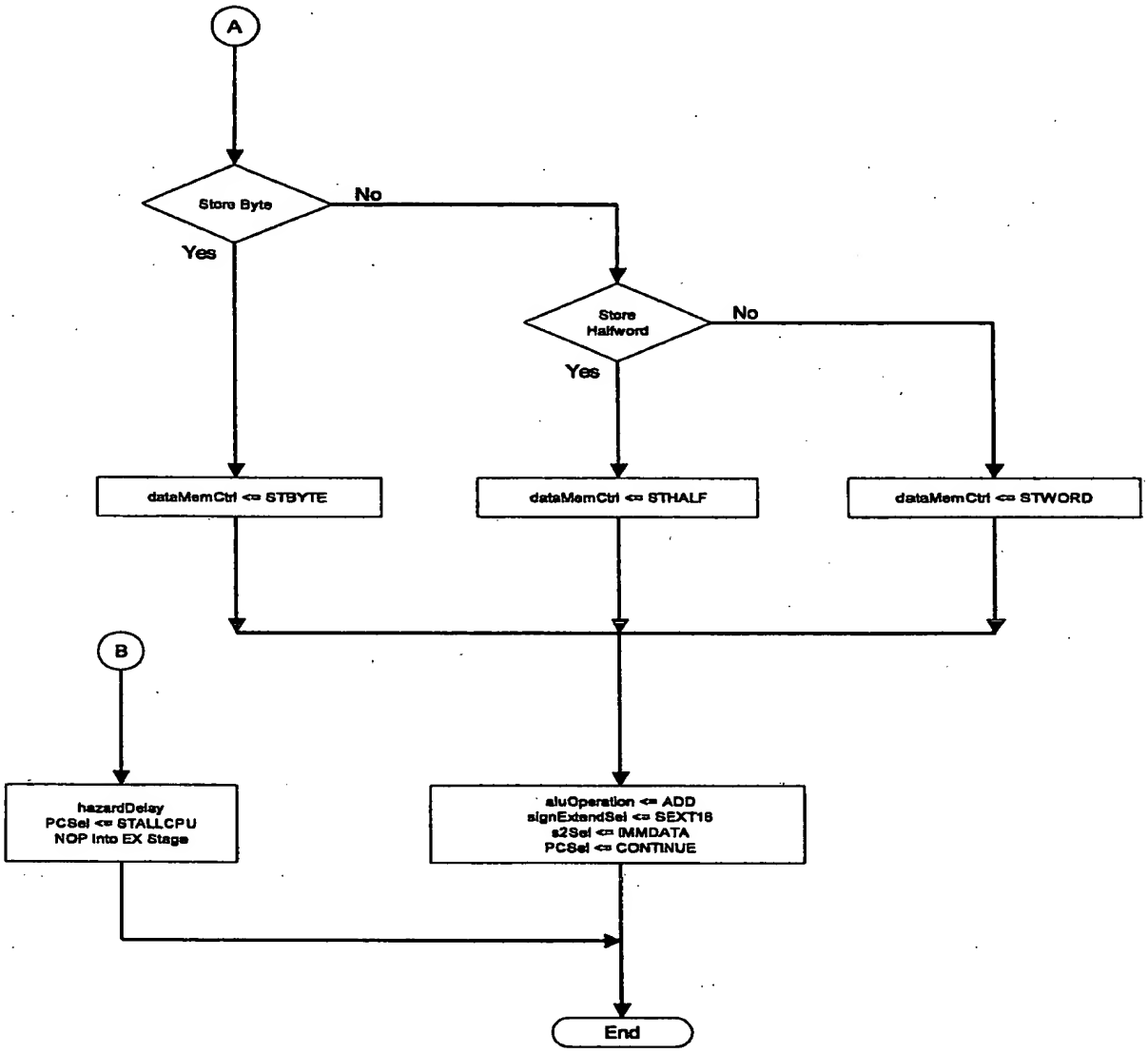

A
Store Byte
No
Yes
Store Halfword
No
Yes
dataMemCtrl <= STBYTE
dataMemCtrl <= STHALF
dataMemCtrl <= STWORD
B
hazardDelay
PCSel <= STALLCPU
NOP into EX Stage
aluOperation <= ADD
signExtendSel <= SEXT16
s2Sel <= IMMDATA
PCSel <= CONTINUE
End

Fig. 21

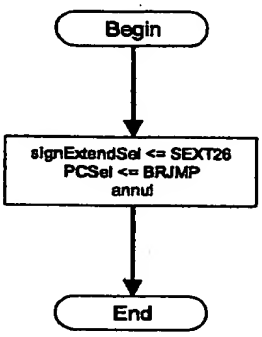

Begin
signExtendSel <= SEXT26
PCSel <= BRJMP
annul
End

Fig. 22

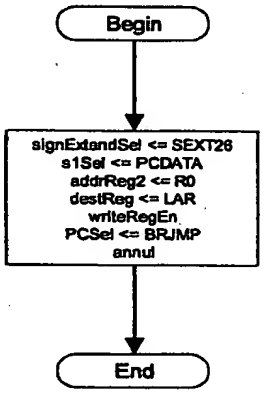

Begin
signExtendSel <= SEXT26
s1Sel <= PCDATA
addrReg2 <= R0
destReg <= LAR
writeRegEn
PCSel <= BRJMP
annul
End

Fig. 23

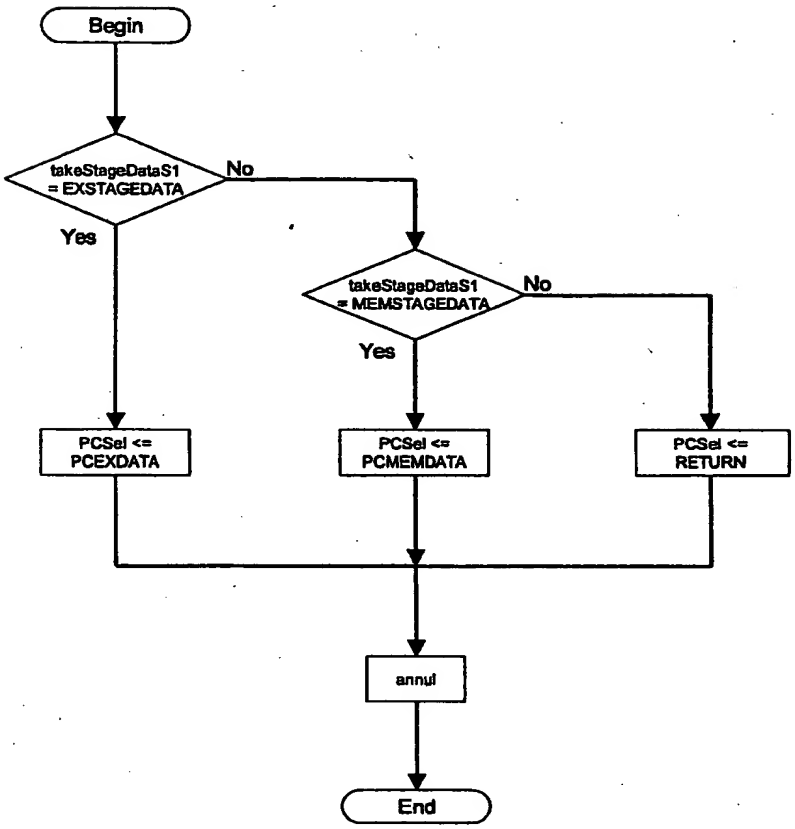

Begin
takeStageDataS1 = EXSTAGEDATA
No
Yes
takeStageDataS1 = MEMSTAGEDATA
No
Yes
PCSel <= PCEXDATA
PCSel <= PCMEMDATA
PCSel <= RETURN
annul
End

Fig. 24

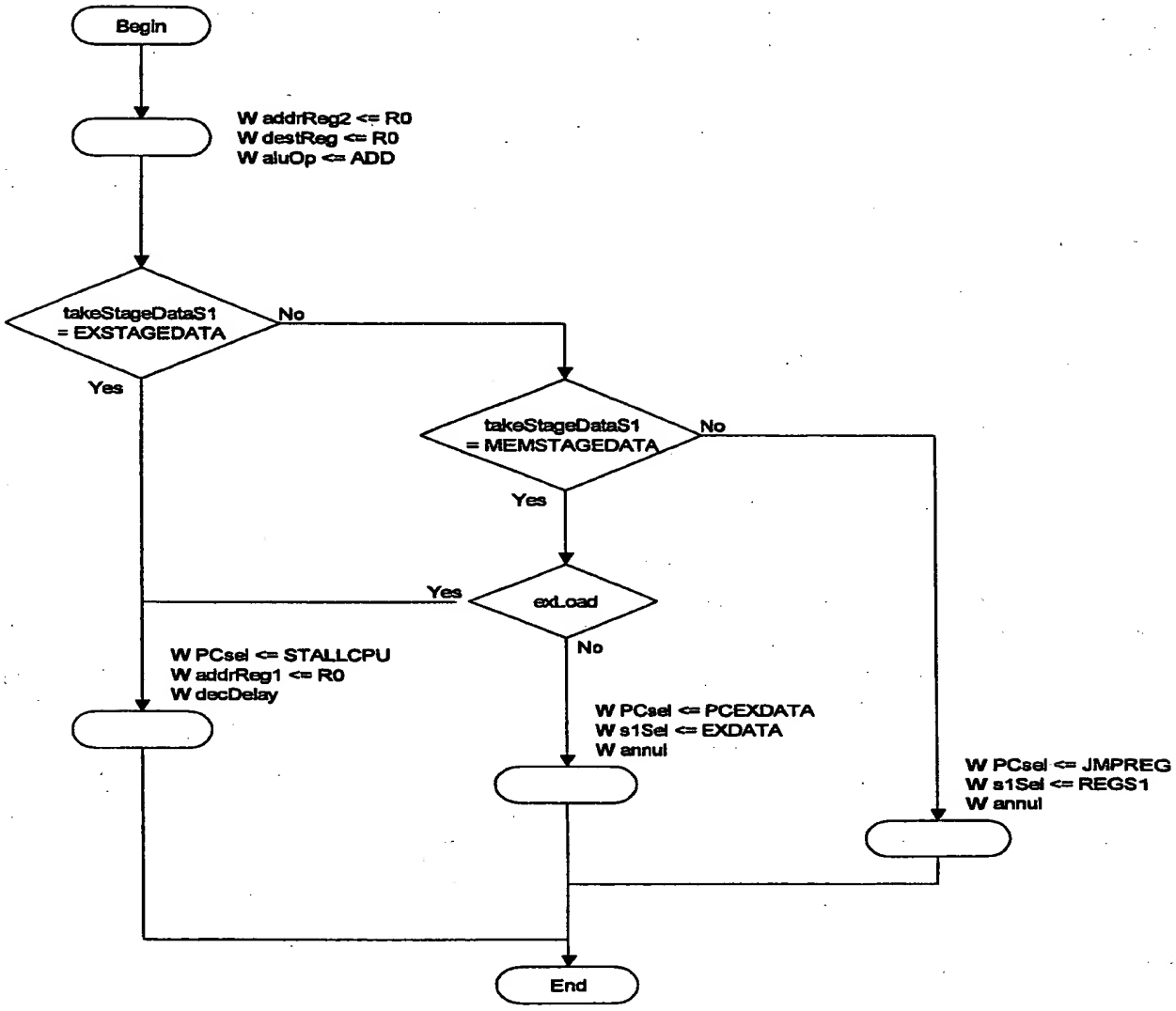

Begin
W addrReg2 <= R0
W destReg <= R0
W aluOp <= ADD
takeStageDataS1
= EXSTAGEDATA
No
Yes
takeStageDataS1
= MEMSTAGEDATA
No
Yes
exLoad
Yes
No
W PCsel <= STALLCPU
W addrReg1 <= R0
W decDelay
W PCsel <= PCEXDATA
W s1Sel <= EXDATA
W annul
W PCsel <= JMPREG
W s1Sel <= REGS1
W annul
End

Fig. 25

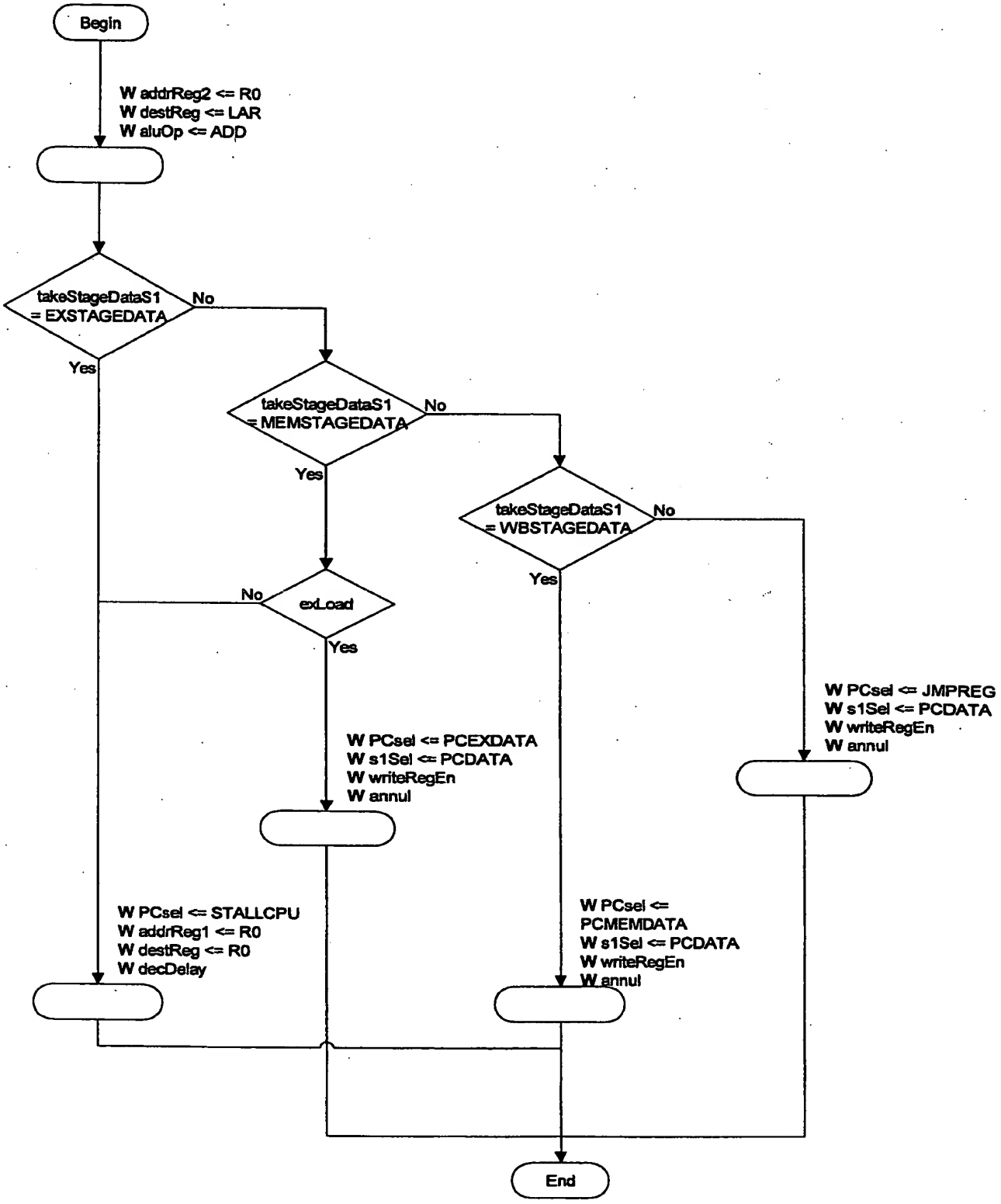

Begin
W addrReg2 <= R0
W destReg <= LAR
W aluOp <= ADD
takeStageDataS1 = EXSTAGEDATA
No
Yes
takeStageDataS1 = MEMSTAGEDATA
No
Yes
takeStageDataS1 = WBSTAGEDATA
No
Yes
exLoad
No
Yes
W PCsel <= JMPREG
W s1Sel <= PCDATA
W writeRegEn
W annul
W PCsel <= PCEXDATA
W s1Sel <= PCDATA
W writeRegEn
W annul
W PCsel <= STALLCPU
W addrReg1 <= R0
W destReg <= R0
W decDelay
W PCsel <= PCMEMDATA
W s1Sel <= PCDATA
W writeRegEn
W annul
End

Fig. 26

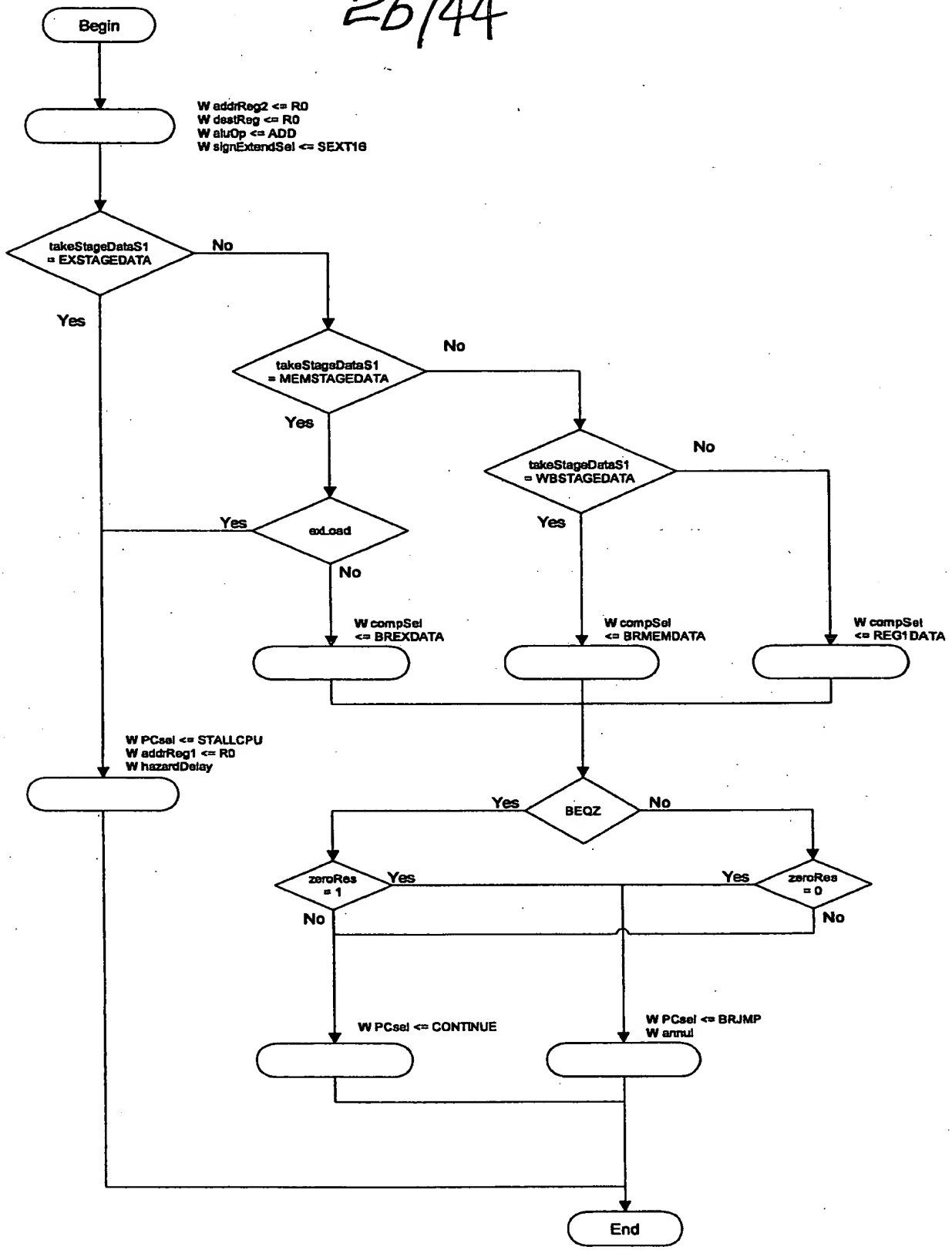

Begin
W addrReg2 <= R0
W destReg <= R0
W aluOp <= ADD
W signExtendSel <= SEXT16
takeStageDataS1 = EXSTAGEDATA
No
Yes
takeStageDataS1 = MEMSTAGEDATA
No
Yes
takeStageDataS1 = WBSTAGEDATA
No
Yes
exLoad
Yes
No
W compSel <= BREXDATA
W compSel <= BRMEMDATA
W compSel <= REG1DATA
W PCsel <= STALLCPU
W addrReg1 <= R0
W hazardDelay
BEQZ
Yes
No
zeroRes = 1
Yes
No
zeroRes = 0
Yes
No
W PCsel <= CONTINUE
W PCsel <= BRJMP
W annul
End

Fig. 27

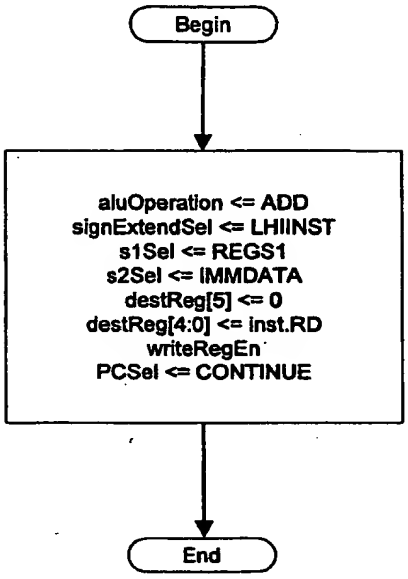


Fig. 28

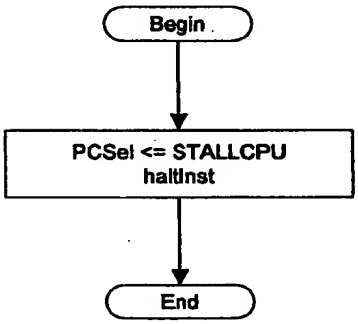


Fig. 29

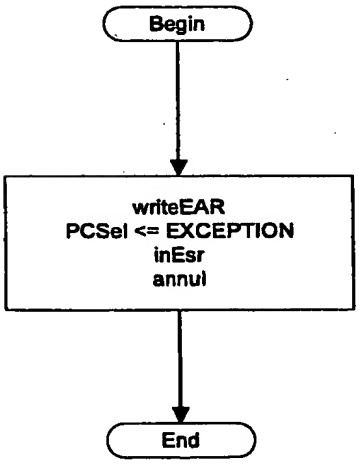


Fig. 30

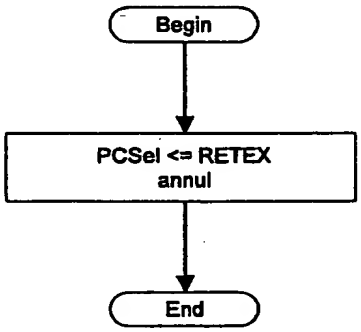

Begin
PCSel <= RETEX
annul
End

Fig. 31

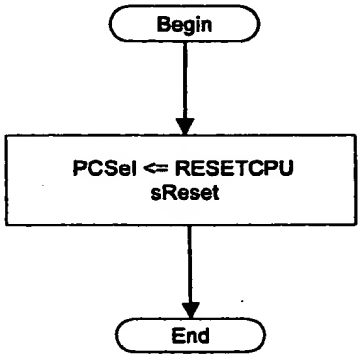

Begin
PCSel <= RESETCPU
sReset
End

Fig. 32

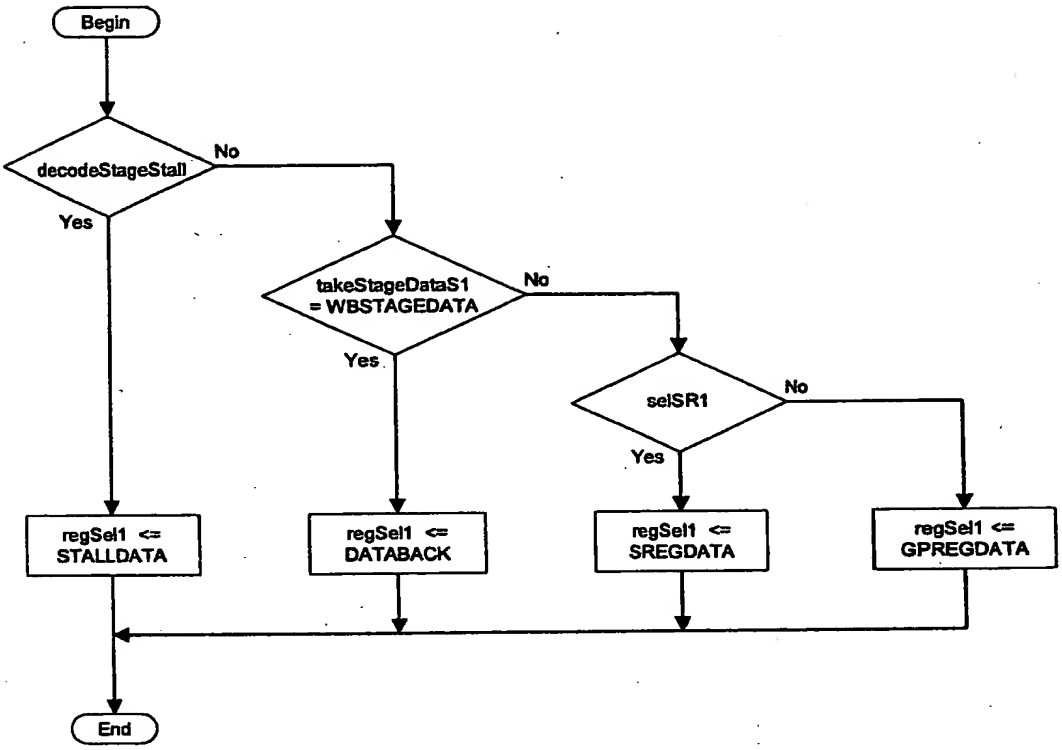

Begin
decodeStageStall
No
Yes
takeStageDataS1
= WBSTAGEDATA
No
Yes
selSR1
No
Yes
regSel1 <=
STALLDATA
regSel1 <=
DATABACK
regSel1 <=
SREGDATA
regSel1 <=
GPREGDATA
End

Fig. 33

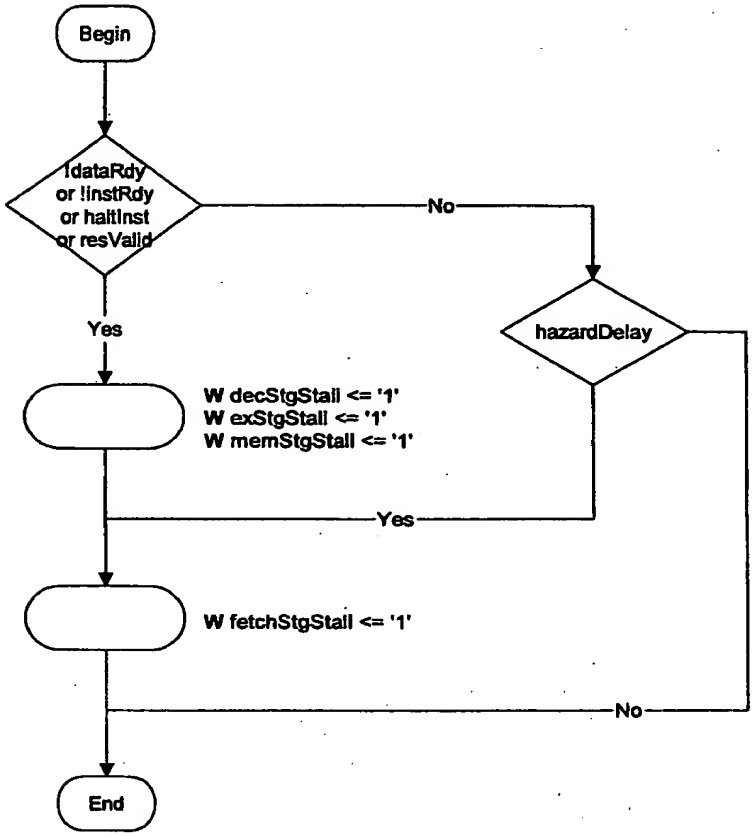

Begin
!dataRdy
or !instRdy
or haltInst
or resValid
No
Yes
hazardDelay
W decStgStall <= '1'
W exStgStall <= '1'
W memStgStall <= '1'
Yes
W fetchStgStall <= '1'
No
End

Fig. 34

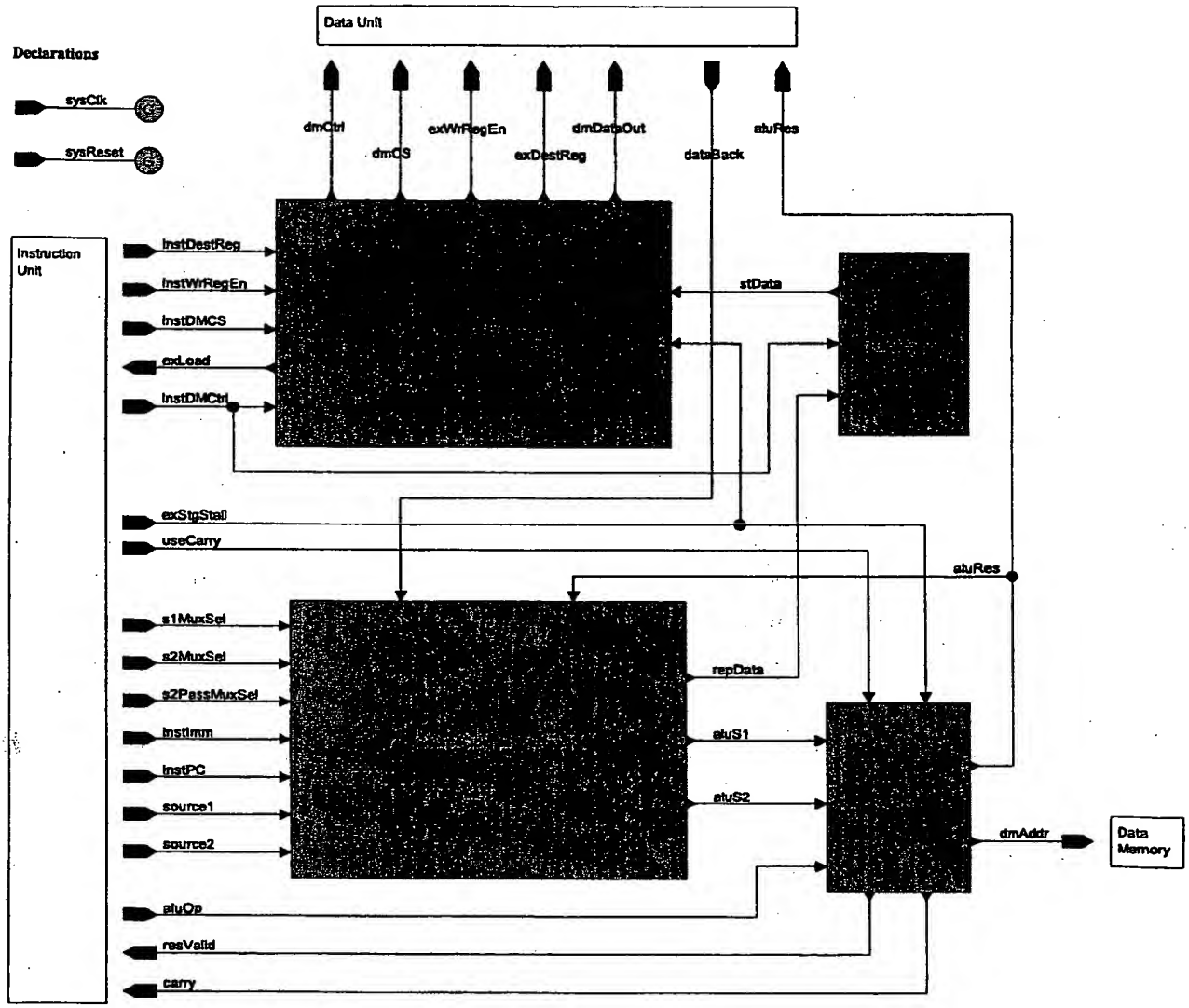

Declarations
sysClk
sysReset
Data Unit
dmCtrl
dmCS
exWrRegEn
exDestReg
dmDataOut
dataBack
aluRes
Instruction Unit
instDestReg
instWrRegEn
instDMCS
exLoad
instDMCtrl
stData
exStgStall
useCarry
aluRes
s1MuxSel
s2MuxSel
s2PassMuxSel
instImm
instPC
source1
source2
repData
aluS1
aluS2
dmAddr
Data Memory
aluOp
resValid
carry

Fig. 35

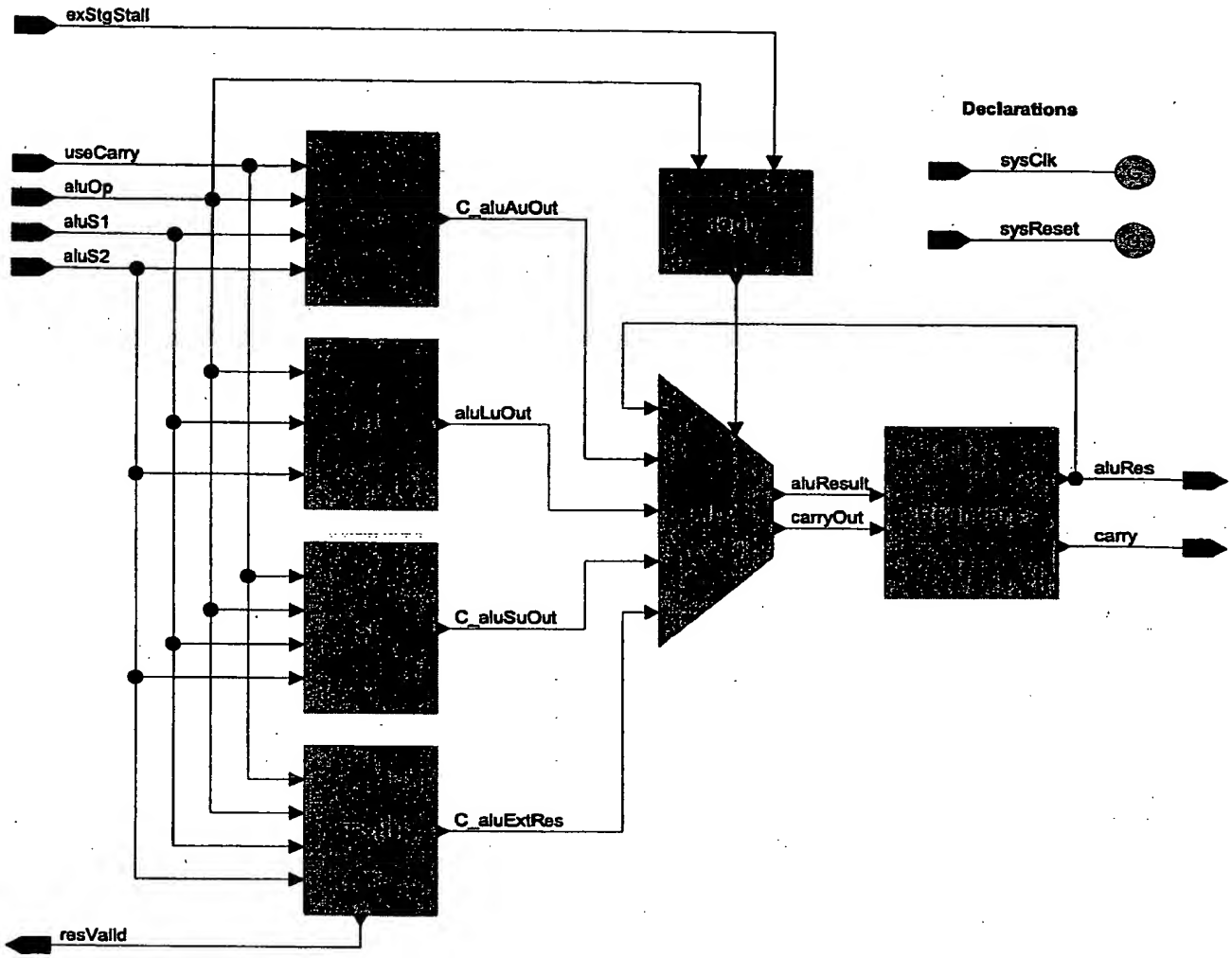

exStgStall
useCarry
aluOp
aluS1
aluS2
C_aluAuOut
aluLuOut
C_aluSuOut
C_aluExtRes
resValid
aluResult
carryOut
aluRes
carry
Declarations
sysClk
sysReset

Fig. 36

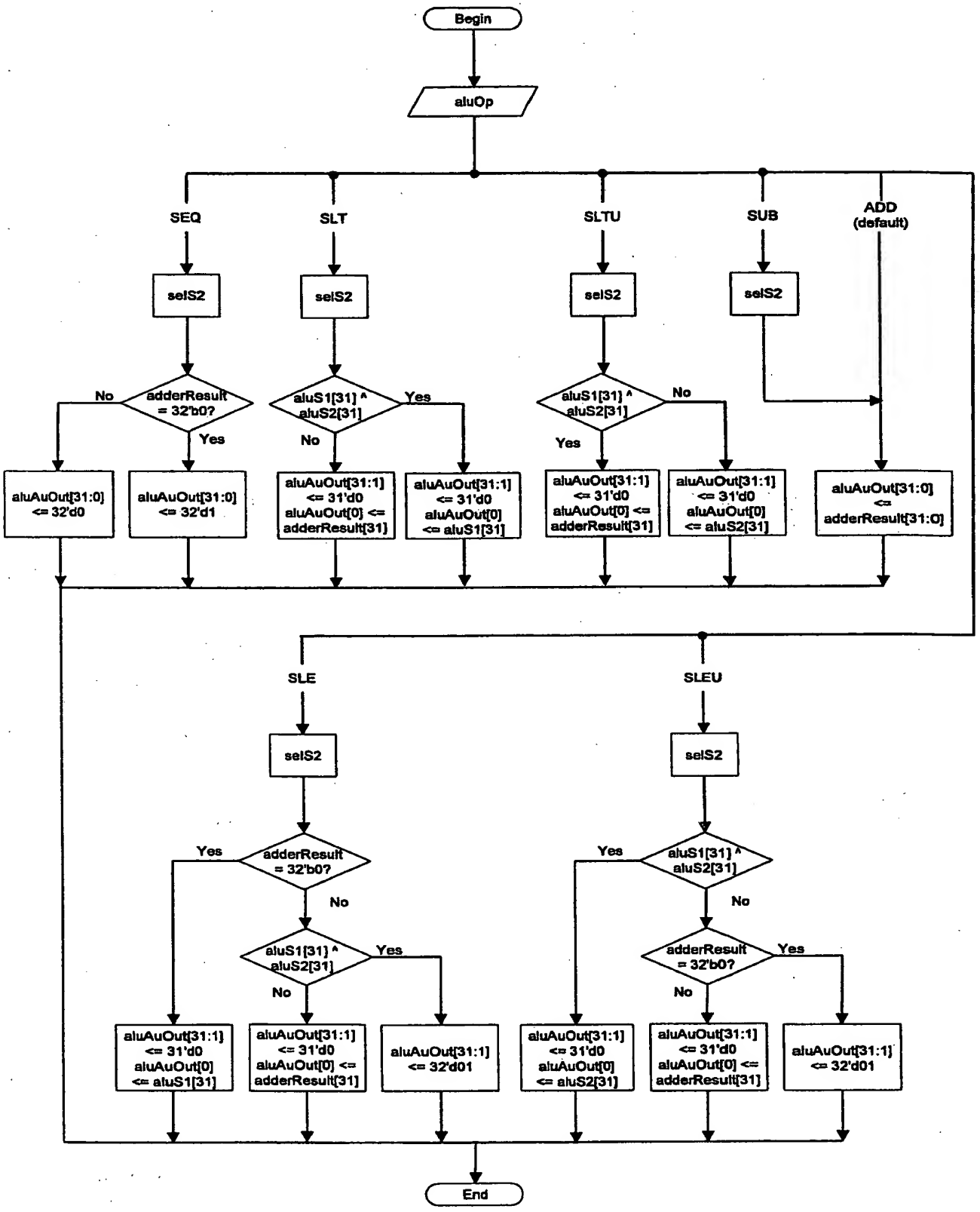

Begin
aluOp
SEQ
SLT
SLTU
SUB
ADD (default)
selS2
selS2
selS2
selS2
No
adderResult = 32'b0?
Yes
aluS1[31] ^ aluS2[31]
Yes
No
aluS1[31] ^ aluS2[31]
No
Yes
aluAuOut[31:0] <= 32'd0
aluAuOut[31:0] <= 32'd1
aluAuOut[31:1] <= 31'd0 aluAuOut[0] <= adderResult[31]
aluAuOut[31:1] <= 31'd0 aluAuOut[0] <= aluS1[31]
aluAuOut[31:1] <= 31'd0 aluAuOut[0] <= adderResult[31]
aluAuOut[31:1] <= 31'd0 aluAuOut[0] <= aluS2[31]
aluAuOut[31:0] <= adderResult[31:0]
SLE
SLEU
selS2
selS2
Yes
adderResult = 32'b0?
No
aluS1[31] ^ aluS2[31]
Yes
No
Yes
aluS1[31] ^ aluS2[31]
No
adderResult = 32'b0?
Yes
No
aluAuOut[31:1] <= 31'd0 aluAuOut[0] <= aluS1[31]
aluAuOut[31:1] <= 31'd0 aluAuOut[0] <= adderResult[31]
aluAuOut[31:1] <= 32'd01
aluAuOut[31:1] <= 31'd0 aluAuOut[0] <= aluS2[31]
aluAuOut[31:1] <= 31'd0 aluAuOut[0] <= adderResult[31]
aluAuOut[31:1] <= 32'd01
End

Fig. 37

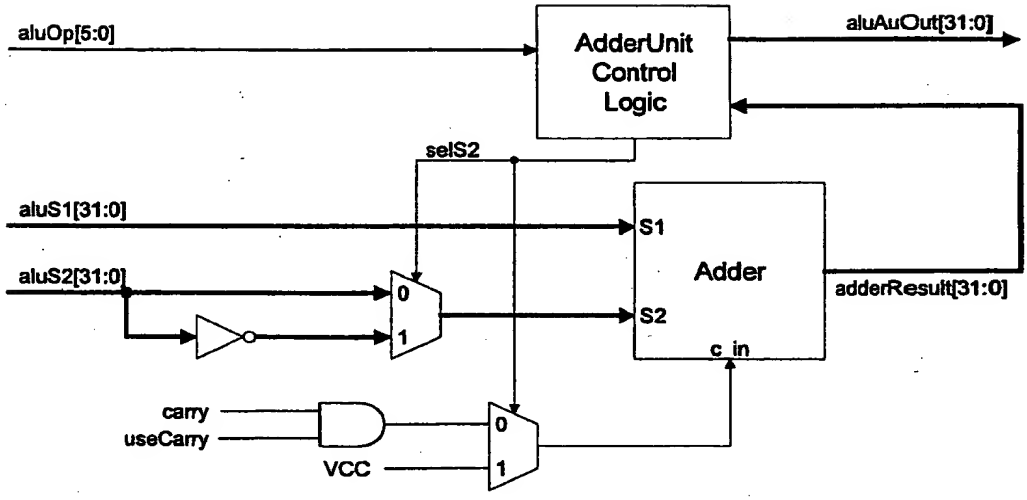

aluOp[5:0]
AdderUnit
Control
Logic
aluAuOut[31:0]
selS2
aluS1[31:0]
S1
Adder
aluS2[31:0]
0
1
S2
adderResult[31:0]
c_in
carry
useCarry
VCC
0
1

Fig. 38

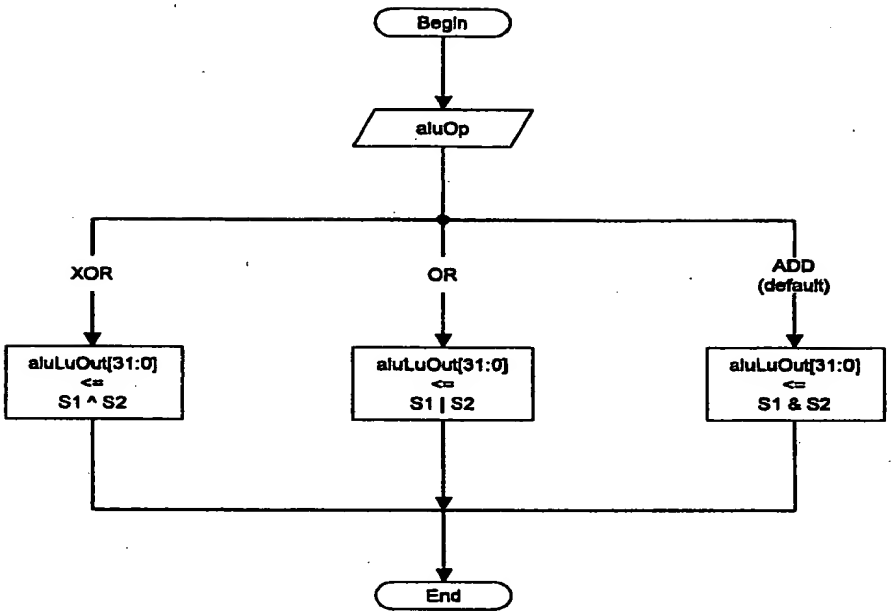

Begin
aluOp
XOR
OR
ADD
(default)
aluLuOut[31:0]
<=
S1 ^ S2
aluLuOut[31:0]
<=
S1 | S2
aluLuOut[31:0]
<=
S1 & S2
End

Fig. 39

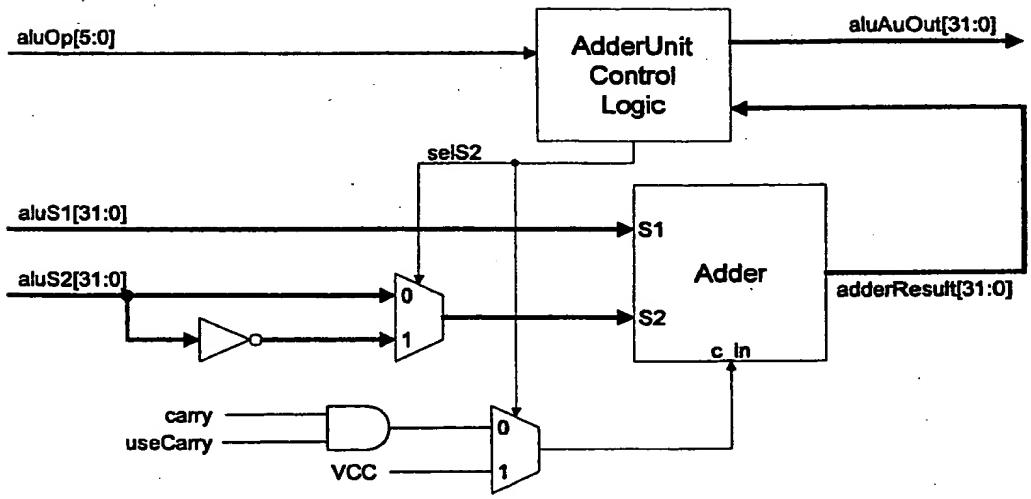

aluOp[5:0]
AdderUnit
Control
Logic
aluAuOut[31:0]
selS2
aluS1[31:0]
S1
Adder
aluS2[31:0]
0
1
S2
adderResult[31:0]
c_in
carry
useCarry
VCC
0
1

Fig. 40

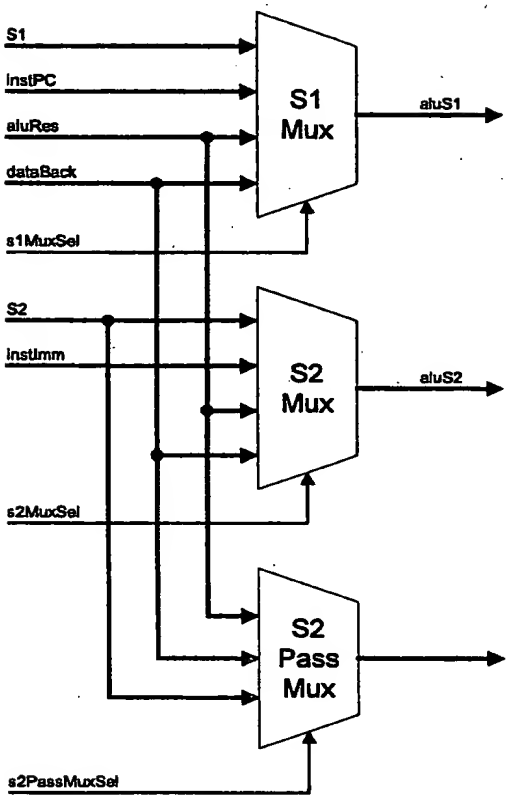

S1
instPC
aluRes
dataBack
s1MuxSel
S1
Mux
aluS1
S2
instImm
s2MuxSel
S2
Mux
aluS2
S2
Pass
Mux
s2PassMuxSel

Fig. 41

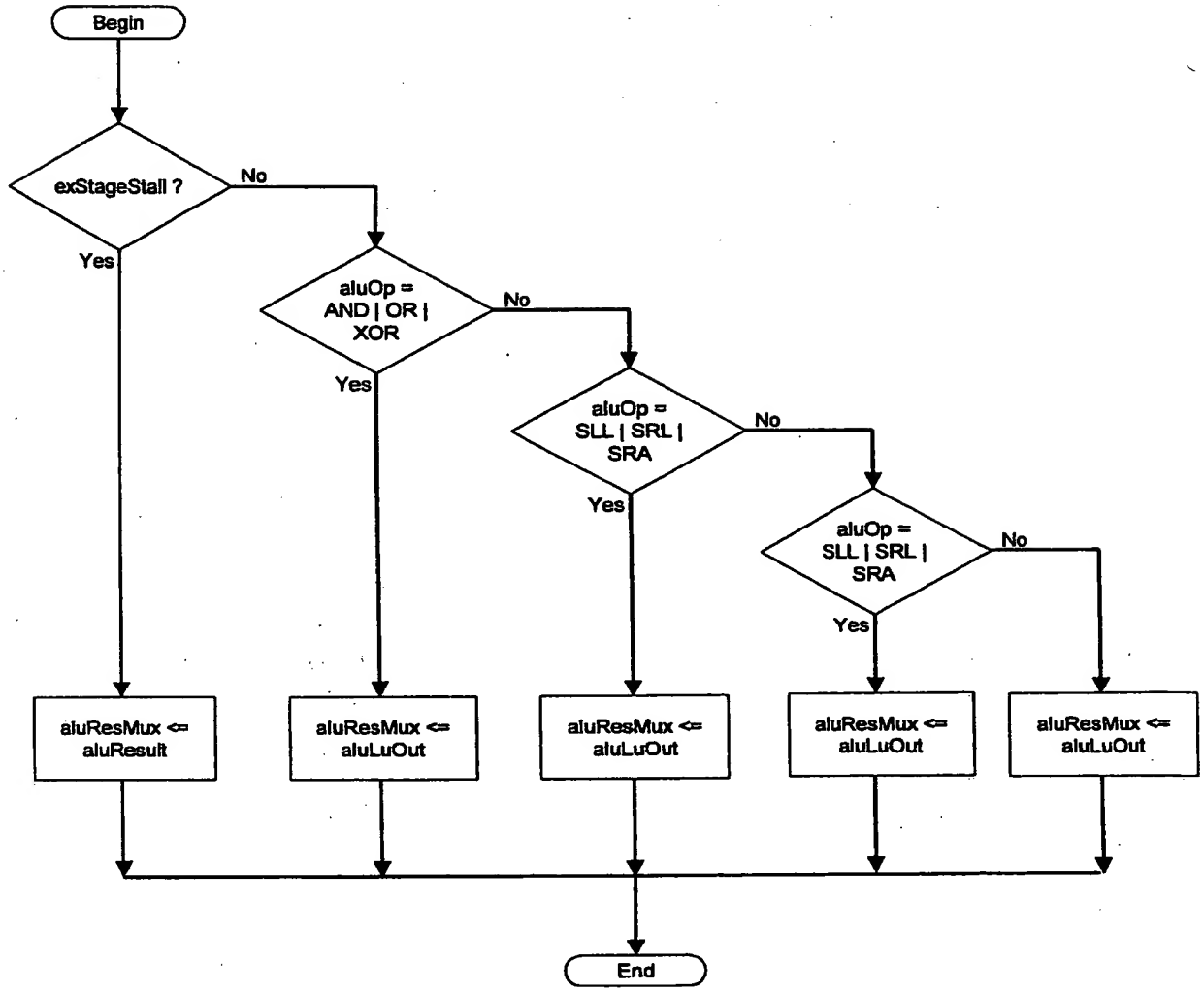

Begin
exStageStall ?
No
Yes
aluOp = AND | OR | XOR
No
Yes
aluOp = SLL | SRL | SRA
No
Yes
aluOp = SLL | SRL | SRA
No
Yes
aluResMux <= aluResult
aluResMux <= aluLuOut
aluResMux <= aluLuOut
aluResMux <= aluLuOut
aluResMux <= aluLuOut
End

Fig. 42

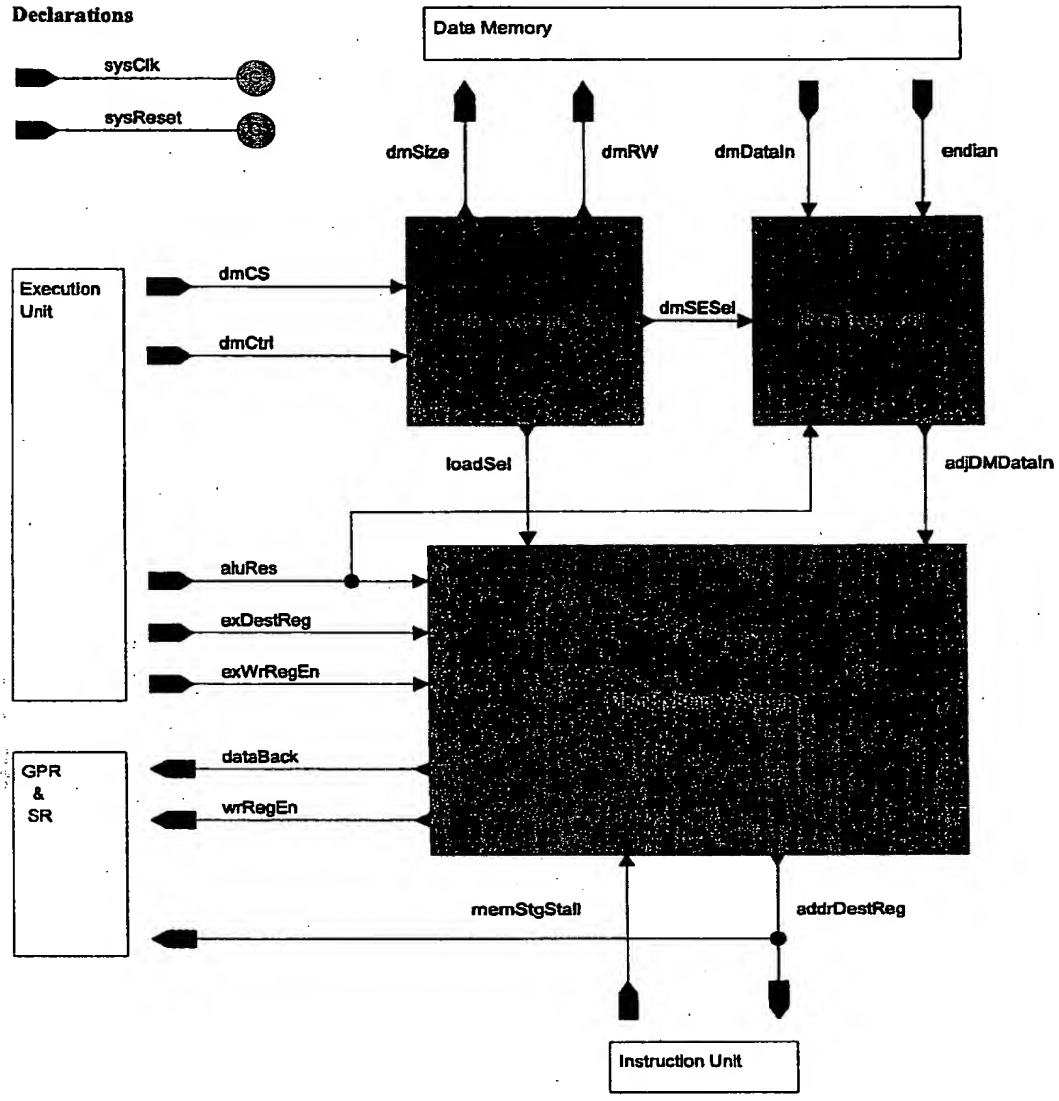

Declarations
sysClk
sysReset
Data Memory
dmSize
dmRW
dmDataIn
endian
Execution Unit
dmCS
dmCtrl
dmSESel
loadSel
adjDMDataIn
aluRes
exDestReg
exWrRegEn
GPR & SR
dataBack
wrRegEn
memStgStall
addrDestReg
Instruction Unit

Fig. 43

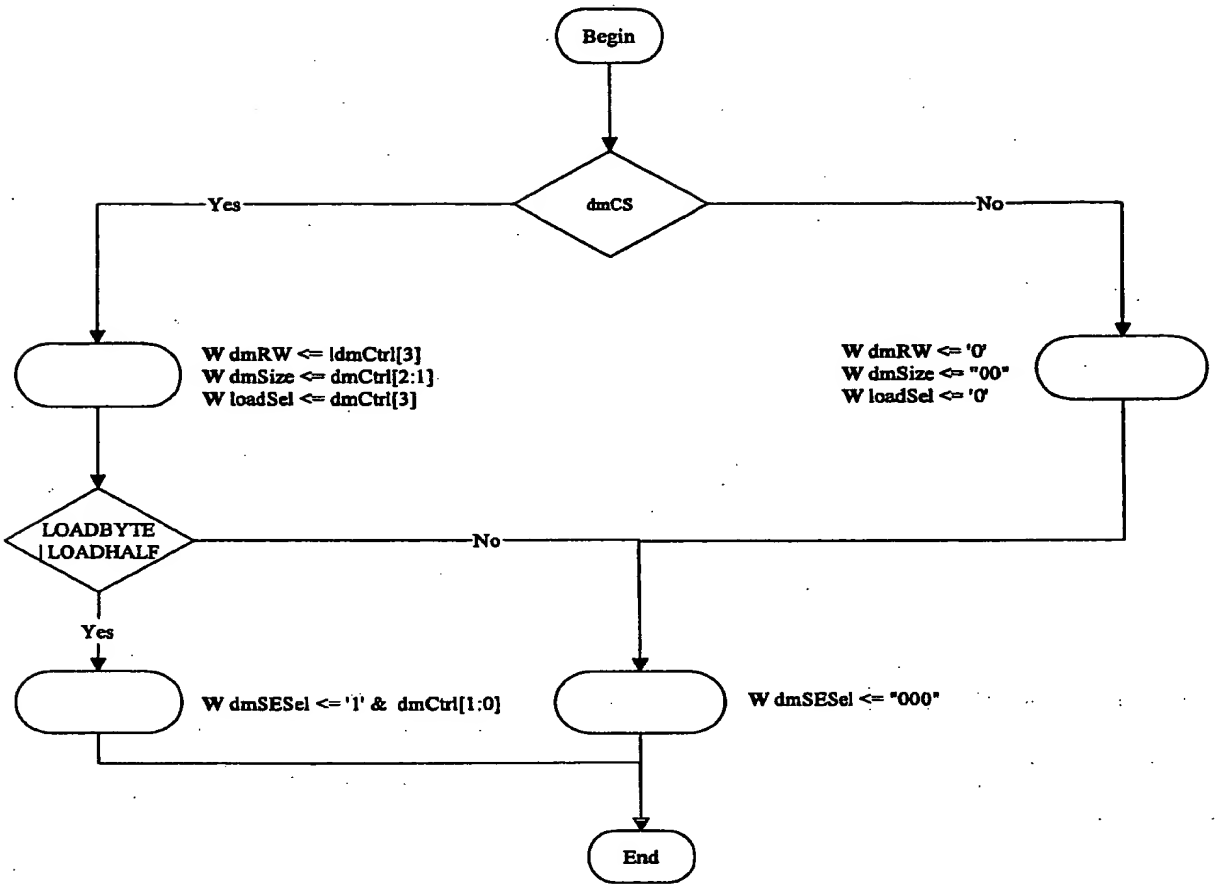

Begin
dmCS
Yes
No
W dmRW <= !dmCtrl[3]
W dmSize <= dmCtrl[2:1]
W loadSel <= dmCtrl[3]
W dmRW <= '0'
W dmSize <= "00"
W loadSel <= '0'
LOADBYTE
| LOADHALF
No
Yes
W dmSESel <= '1' & dmCtrl[1:0]
W dmSESel <= "000"
End

Fig. 44

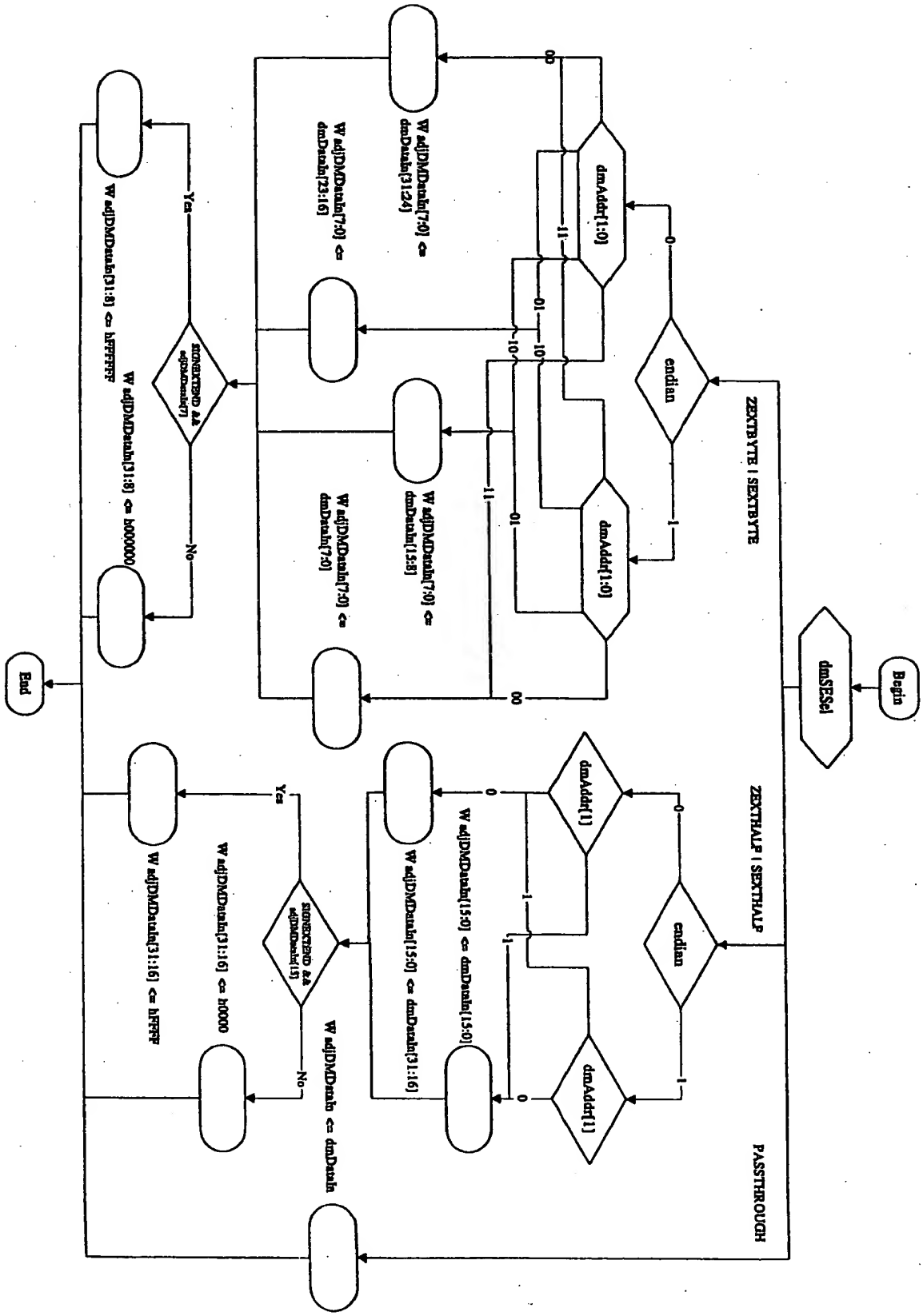

Begin
dmSESel
ZEXTBYTE | SEXTBYTE
ZEXTHALF | SEXTHALF
PASSTHROUGH
endian
endian
dmAddr[1:0]
dmAddr[1:0]
dmAddr[1]
dmAddr[1]
W edjDMDatalo[7:0] <= dmData[31:24]
W edjDMDatalo[7:0] <= dmData[23:16]
W edjDMDatalo[7:0] <= dmData[15:8]
W edjDMDatalo[7:0] <= dmData[7:0]
W edjDMDatalo[15:0] <= dmData[15:0]
W edjDMDatalo[15:0] <= dmData[31:16]
W edjDMDatalo <= dmData
SIGNEXTEND && edjDMDatalo[7]
SIGNEXTEND && edjDMDatalo[15]
Yes
No
W edjDMDatalo[31:8] <= hFFFFFF
W edjDMDatalo[31:8] <= h000000
W edjDMDatalo[31:16] <= hFFFF
W edjDMDatalo[31:16] <= h0000
End

Fig. 45

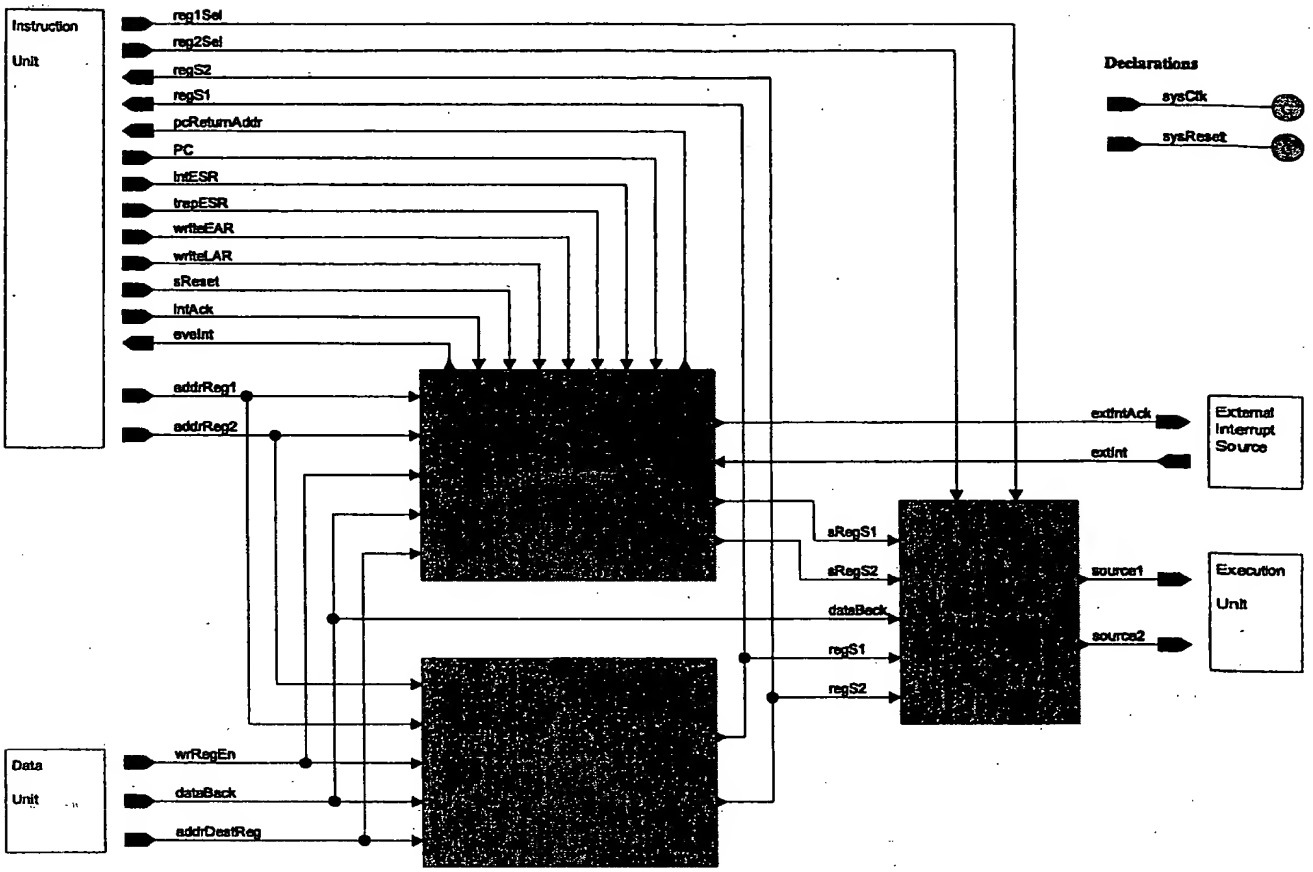

Instruction
Unit
reg1Sel
reg2Sel
regS2
regS1
pcReturnAddr
PC
intESR
trapESR
writeEAR
writeLAR
sReset
intAck
eveInt
addrReg1
addrReg2
Declarations
sysClk
sysReset
extIntAck
extInt
External
Interrupt
Source
sRegS1
sRegS2
dataBack
regS1
regS2
source1
source2
Execution
Unit
Data
Unit
wrRegEn
dataBack
addrDestReg

Fig. 46

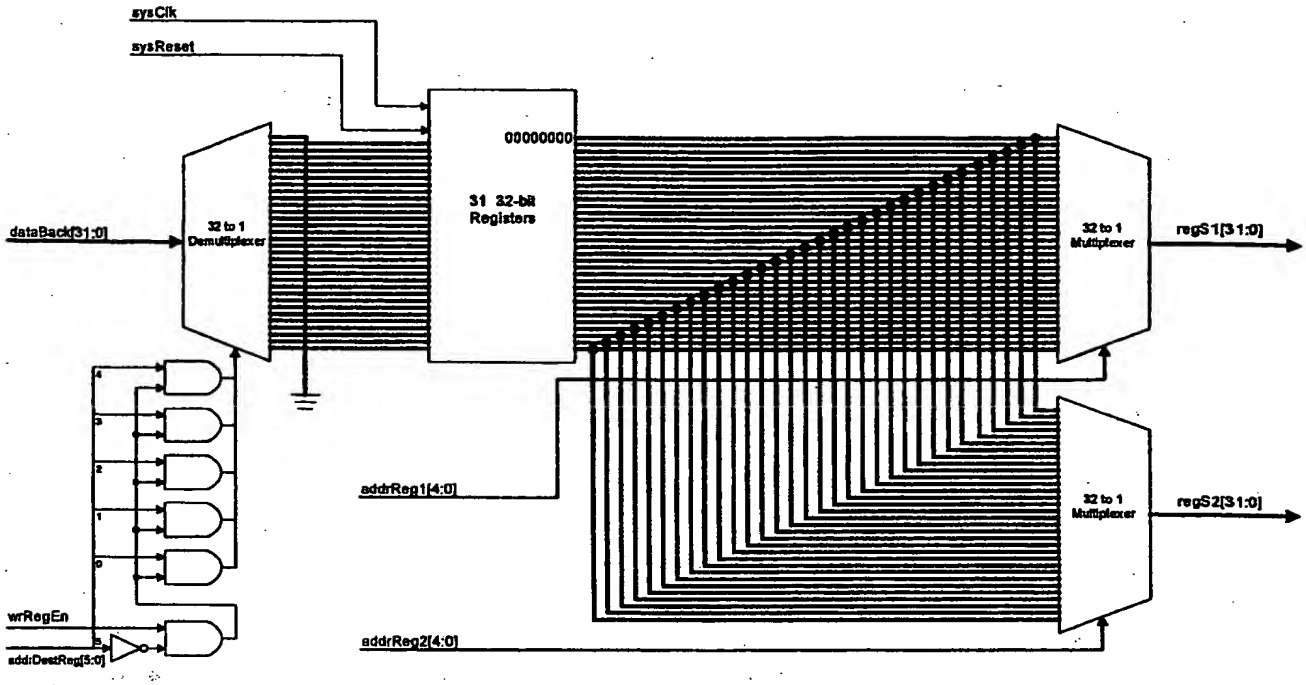

sysClk
sysReset
00000000
31 32-bit
Registers
dataBack[31:0]
32 to 1
Demultiplexer
32 to 1
Multiplexer
regS1[31:0]
addrReg1[4:0]
32 to 1
Multiplexer
regS2[31:0]
wrRegEn
addrDestReg[5:0]
addrReg2[4:0]

Fig. 47

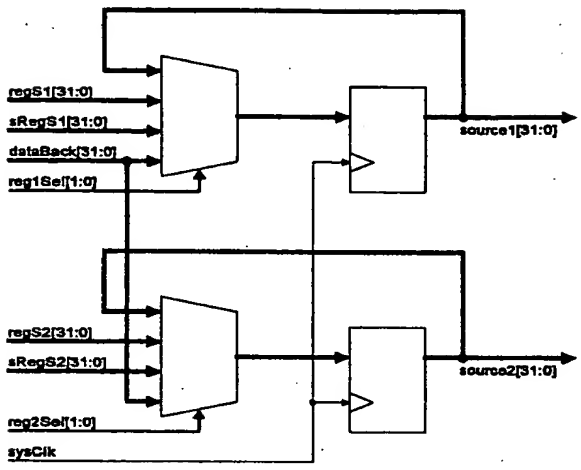

regS1[31:0]
sRegS1[31:0]
dataBack[31:0]
reg1Sel[1:0]
source1[31:0]
regS2[31:0]
sRegS2[31:0]
reg2Sel[1:0]
sysClk
source2[31:0]

Fig. 48